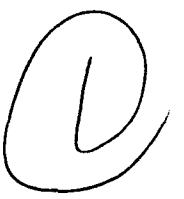
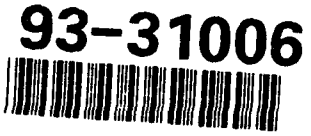


GRAPHICAL TOOLS FOR SITUATIONAL AWARENESS ASSISTANCE FOR LARGE BATTLE SPACES

THESIS

Brian B. Soltz, Capt, USAF
AFIT/GCS/ENG-93D-21



AFIT/GCS/ENG/93D-21

GRAPHICAL TOOLS FOR SITUATIONAL AWARENESS ASSISTANCE FOR LARGE SYNTHETIC BATTLE SPACES

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Science

Brian B. Soltz

Captain, USAF

December 1993

Approved for public release; distribution unlimited

PREFACE

I am indebted to several people for their assistance and cooperation with this project. In particular, I would like to thank Capt Kirk Wilson, for his insight in the design and implementation of the Synthetic BattleBridge that made integrating my system simple and straight forward. There are also a number of other people who I would like to thank for their support and comradely: Mr Steve Sheasby, Mr Dave Doak, Capt Matthew Erichsen, Maj Michael Gardner, Capt William Gerhard, Capt Allain Jones, Capt Andrea Kunz, Capt Keith Meissner, and Capt Mark Snyder. I would also like to thank LtCol Martin Stytz and LtCol Phil Amburn for their guidance and understanding over the last hard five months.

I especially wish to thank my wife, Marie, and my daughter, Jennifer, for their extreme patience, understanding, and love. Without their support, I would not have been able to go as far as I did, they were my inspiration.

Brian B. Soltz

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

As virtual environments grow in complexity and size, users are increasingly challenged in assessing situations in large-scale virtual environment. This occurs because of the difficulty in determining where to focus attention and assimilating and assessing the information as it floods in. One technique for providing this type of assistance is to provide the user with a first-person, immersive, synthetic environment observation post, that permits unobtrusive observation of the environment without interfering with the activity in the environment. However, for large, complex synthetic environments, this type of support is not sufficient because the portrayal of raw, unanalyzed data in the virtual space can overwhelm the user. To address these problems, this thesis investigates the types of situational awareness assistance that need to be provided to users of large-scale virtual environments. A technique developed, is to allow a user to place analysis modules throughout the virtual environment. Each module provides summary information to the user concerning the status of the section of the virtual environment that the module was assigned to monitor. The prototype system, called the Sentinel, is embedded within a virtual environment observatory and provides situational awareness assistance for users within a large virtual environment.

GRAPHICAL TOOLS FOR SITUATIONAL AWARENESS ASSISTANCE FOR LARGE SYNTHETIC BATTLE SPACES

I. INTRODUCTION

1.1 Background

With the end of the Cold War a few years ago, military doctrine has changed dramatically. No longer is the perceived threat that of one large, powerful enemy. Instead, due to the breakup of the Soviet Union block, many alliances have fallen to the wayside which has renewed many long standing religious and ethnic hatreds that were subdued in the old Soviet regime. Other political and economic problems throughout the world have also caused many areas of unrest that could also be a potential threat to US and allied interests.

Planners, both military and civilian, develop scenarios to plan for these threats. They are forced to consider the use of armed forces covering the spectrum from full scale war such as Desert Shield and Desert Storm to humanitarian relief efforts like those used in Somalia. This unprecedented use of armed forces coupled with decreasing defense expenditures, force military leaders to seek innovative and economical alternatives to tactical battlefield analysis, mission planning, and training systems. One area receiving increased attention is the use of virtual reality or synthetic environments.

One such synthetic environment viewer, the Synthetic BattleBridge (SBB), allows the user to view the battlefield as a passive observer. This works fine for small scale simulations as the user can observe almost everything in the battlefield. However, when the simulation addresses a large scale battlefield, the user can see only what the simulation can show them for the specific current view. If the user tries to back away to see the whole

battlefield, then the resolution of the objects in the simulation become blurred and distorted. Therefore, the user can not make valid analysis of the risk assessment for the current simulation.

The answer to such a problem is to have another system globally watch over the entire simulation and provide situational awareness assistance for the user of the current simulation. This other system can then inform the user of other areas (watchspaces) with moderate or high risk, that they might want to view. Therefore, the other system can take the burden of watching the entire simulation off the user and allow them to concentrate on specific areas within the simulation. A Fuzzy Controller System, known as the Sentinel, accomplishes this situational awareness assistance for the user of the overall battlefield simulation.

1.2 Problem Statement

Design and implement a Fuzzy Controller System (Sentinel) to perform situational awareness assistance of pre-defined areas (watchspaces) within a synthetic environment. The particular synthetic environment targeted is the SBB. The Fuzzy Controller (Sentinel) and support subsystems convey situational awareness information to the user with the efficient use of on-screen displays and sound cues. The on-screen displays use a variety of colors to quickly let the user make an assessment of risk for each of their pre-defined watchspaces of interest. The Fuzzy Controller System (Sentinel) works in the background while the SBB processes and views the current simulation. Therefore, the user uses the Fuzzy Controller (Sentinel) as a situational awareness tool that can be turned on or off as needed.

1.3 Summary Of Current Knowledge

This thesis topic ties together three areas of research: the use of virtual reality and synthetic environments to allow the user an immersive view of a simulation, the use of fuzzy logic as a situational awareness tool, and the use of human/computer interface techniques to enhance overall system usability.

The first area uses virtual reality and synthetic environments to provide a user with a three-dimensional representation of moving and stationary vehicles dispersed over a large area of terrain. The environment accommodates increasing levels of resolution for both the terrain and the vehicles of interest and provides the user with an intuitive and modifiable interface. The SBB developed by Capt. Rex Haddix in 1992 addresses many of the issues involved with a synthetic environment. Current work by Capt. Kirk Wilson ([Wil93]) and I continues where Capt. Rex Haddix left off by making the SBB easier to use, more capable, and technically applicable to users in the field.

The second area deals with using fuzzy logic to develop a system that can make situational awareness judgments based on predetermined risk categories. Fuzzy logic can do risk analysis based on variables that are conceptually vague in nature. Current papers describe how fuzzy logic combines the uncertainty of given variables with user studies that indicate what actions would actually take place. Using this information, we can design a Fuzzy Controller (Sentinel) that for a given set of fuzzy inputs, it produces a single fuzzy output. The fuzzy output is then attached to a color code and bar length that is viewed by the user. The user then mentally converts this color and length information into a relative assessment of the risk or activity in that watchspace. In other words, we can take actual numbers, perform fuzzy set operations on them to produce a single relative number, that the user can process in terms relative to themselves.

The third area, human/computer interface techniques, ties together the first two areas mentioned above. How we display information to the user depends upon the type

and importance of the information. Visual and sound cues instantly give the user information needed about the current state of the system. Determining what visual and sound cues to use is the key to portraying information in a timely and efficient manner.

1.4 Scope

This thesis is the first attempt to apply fuzzy logic to a synthetic environment to enhance situational awareness for the user. It is limited in capabilities and is primarily being used as a proof of concept for the theory and techniques involved with fuzzy control and situational awareness. Additionally, the Fuzzy Controller (Sentinel) itself depends upon limited user studies to fine tune the system. However, this thesis shows the practical application of the Fuzzy Controller System (Sentinel) to an actual synthetic environment, namely the SBB.

1.5 Approach and Methodology

The approach taken for the design and implementation of the Fuzzy Controller (Sentinel) to the SBB, breaks up the Fuzzy Controller System (Sentinel) into a number of distinct independent modules. Each one of these modules is then designed, tested, and implemented separately using pre-defined interface specifications between modules. Finally, the integration of all the modules together takes place along with testing the system as a whole. These modules are then encapsulated from the user by the use of object-oriented classes that ties together the whole system.

The Fuzzy Controller System (Sentinel) is composed of five distinct modules. The first module, the Configuration Unit, takes in and translates user configuration information into the configuration data structure used by the system. The second module, the Input Unit, uses the configuration data structure along with current object information about the

simulation to calculate the fuzzy input parameters needed by the Fuzzy Logic Computation Unit. This information feeds into the third module, the Fuzzy Logic Computation Unit, which does the actual risk or activity assessment of the fuzzy parameters for each pre-defined watchspace of interest. This information passes to the fourth module, the Output Unit, that processes the watchspace assessment value received for each watchspace and visually displays this information on the screen to the user. This module also takes care of any interrupt handling that needs to go on based on watchspace assessments. The fifth module, the Control Unit, is the actually interface between the user and the Fuzzy Logic (Sentinel) watchspaces.

The nature of the Output Unit module indicates that further subdivision of the module could take place. This subdivision takes into account the various different displays that are needed to convey information to the user. We also need to take into account the various sound cues required by the system. Overall, the Output Unit becomes the vital link between the Fuzzy Controller System (Sentinel), the synthetic environment system (SBB), and the user.

The classes play an important role in encapsulating the design and implementation from the user. There are two classes that make up the Sentinel system. The first class, the FL_Sentinel Class, is the main class that manages all the overhead and communication between the modules (units), the FLS_Player Class, and the user. The second class, the FLS_Player Class, handles all the rendering issues associated with the Sentinel watchspaces. It is the FLS_Player Class that the Control Unit mentioned above has the most impact on. User commands are interpreted from the Control Unit, communicated to the FLS_Player Class through the FL_Sentinel Class, and then acted upon by the FLS_Player Class that in turn displays the effects of the user selected function.

1.6 Materials and Equipment

The implementation of the Sentinel using fuzzy logic set theory requires that the extra computational power needed to do the fuzzy logic does not slow down the rendering process. If the use of the Sentinel system causes a significant drop in frame rate, then the overall driving system, the SBB in this case, becomes less of a time analytical tool and more of tool that performs analysis on the simulation data. In other words, if we wish to use the Sentinel as a situational awareness tool to help the user make command decisions about the current state of the simulation, then it must present information to the user in as close to real time as possible without distortion.

There are many things that the Sentinel system introduces into the driving application that could have a profound effect on the frame rate. Possible areas most effecting the overall frame rate are as follows:

- Mathematical computations need by the fuzzy logic computational unit and Sentinel volume watchspace containment functions;
- Graphics rendering pipeline as affected by the display of additional Sentinel user control panels and display of Sentinel watchspace geometric representations;
- Z-buffering algorithm as it pertains to displaying transparent Sentinel watchspaces in such a way as to make then visible by all other transparent objects in the scene;
- User input delays associated with the Sentinel user control panels.

The frame rate issue is of key concern when trying to develop a system that will evolve to having over 8000 objects in the simulation at a time in the near future. Frame rates that are too slow look more like a series of pictures rather then a smooth animated scene. This slowness injects jerkiness into the scene and cause user input to be hampered. To achieve an acceptable frame rate, the Sentinel system, along with its driving application

(SBB), was designed to work with a multiprocessor parallel workstation. There are currently two such workstation types for which the system can be run on:

- The Silicon Graphics IRIS 4D/440VGXT Workstation with two or more processors,
- The Silicon Graphics Onyx RealityEngine^{2TM} Workstation with two or more processors.

Both workstations provide a hardware graphics pipeline and sufficient RS-232 ports for the external devices required.

The software was written in C++. It can be compiled with the AT&T C++, version 2.1 or 3.0.1, compilers. It should also be noted that the two workstations above can be equipped with either the version 4.0.x or 5.x Silicon Graphics Operating System. Once again, the software was implemented with these various workstations, compilers, and operating systems in mind.

1.7 Thesis Organization

The reminder of this document describes the steps taken to create the Sentinel system. Chapter two describes user interactions with virtual environments and how they can be extended through the use of situational awareness tools. Chapter two also talks about the use of Fuzzy Logic Controllers in the decision making process. Chapter three describes the design of the Sentinel system and how it pertains to both structured programming and object-oriented methodologies. Chapter four describes the actual implementation of the Sentinel system with the Synthetic BattleBridge as the driving application. Chapter five discusses results and conclusions of integrating and using the Sentinel system. Appendix I provides a users manual for a brief tutorial on using the Sentinel system with the driving application (SBB). Finally, Appendix II provides a

programmers manual that talks about how to compile, modify, use certain method calls, and change the parameters of the Sentinel system.

II. BACKGROUND

2.1 Introduction

As virtual environments grow in complexity, size, and scope users need assistance in comprehending, assessing, and reacting to the state of the environment. One technique for providing this assistance is to provide the user with a first-person, immersive, synthetic environment observation post, an observatory, that permits unobtrusive observation of the environment without interfering with the activity in the environment. However, for large-scale, complex, rapidly changing environments, such as those that occur when simulating a fire, natural disaster, air traffic control, or a battle, this type of support is insufficient. To address this problem, this thesis investigates the types and forms of situational awareness assistance that should be provided to users in these types of synthetic environments. The prototype system provides situational assistance for users within the large virtual environment that exists within the Advanced Research Projects Agency (ARPA) Distributed Interactive Simulation project ([Tho88])[1].

2.1.1 Fog of War

A battlespace is an excellent driving application for our investigation because of the complexity and uncertainty inherent in the environment. The state of the battlespace is in almost constant flux and the important portions of the battlespace differ from moment to moment. For example, at one instant an important activity might be a reconnaissance event, which could be followed by the beginning of an aerial operation, a ground engagement, a dogfight, or the arrival of a resupply mission. The "interest" value of these

[1]This ARPA project is designed to simulate large battlespaces for the purpose of evaluating weapons and tactics, performing integrated engineering and design, performing top-down decision making and analysis, and evaluation of emerging technologies.

and other potentially important events is not assessed in isolation by the user. Instead, their interest value is judged in relation to other events happening at the same time in the same location as well as events happening at the same time at other locations. Because of the complexity and uncertainty, large staffs and management mechanisms have been developed to assist commanders in assessing the state of a battlespace. However, these mechanisms have not been completely successful, because uncertainty about the state of the battlespace remains. This confusion has been termed the fog of war. The fog arises from two complementary problems, information accuracy and information complexity. Information accuracy is a problem caused by uncertain data, which can arise from deliberate enemy deception, observational error, conflicting data, and errors within the information reporting mechanisms. This thesis does not address the problem of confusion about the state of a battlespace that arises from information accuracy problems. This thesis, however, investigates a means for allowing correct situation diagnosis within an informationally complex environment, which is currently difficult due to the rapidly changing and complex information inherent in the battlefield.

Confusion caused by information complexity occurs because the data about a battlespace, or any other large, thickly populated, active environment such as an airport, a large building fire, or satellites in orbit, is complicated and rapidly changing. Because of these characteristics, the state of the battlespace must be comprehended swiftly and its important aspects grasped quickly. The task is further complicated by the fact that the state continuously changes and the location of important information is unknown from moment to moment. The problem of information complexity can be addressed using techniques for data reduction and analysis that have been proven in the fields of computer graphics and human-computer interaction. To test this hypothesis, this thesis effort intends to develop and evaluate techniques for reducing information complexity and supporting situational assessment within a distributed battlespace synthetic environment (discussed below).

2.1.2 *Distributed Synthetic Battlespace Environment*

Situational assessment assumes the identification of a problem that requires some action. The commander must decide on some course of action based on what occurs in the battlespace. To determine the state of the battlespace, environmental cues, such as radar, infrared, text, and observer sightings, are sampled to obtain a situational assessment, or diagnosis, of the state of the battlespace. An accurate diagnosis requires the perception of a large number of cues, which in turn must be interpreted against a knowledge base in long-term memory to accurately construct a mental model of the situation. To form a mental model of a combat situation, the commander needs to be aware of the disposition and capability of his own forces and the disposition and capability of the enemy forces. However, human characteristics work against this process. In forming the mental model, subtle yet critical aspects of the battlespace may be missed, leading to incorrect decisions. Humans have limits of attention that may cause them to process cues that are not the most relevant ([Sol91]). Nevertheless it is vital for correct decision making that the user process not the most salient symptoms, but instead the most relevant ones because they provide the most important diagnostic information concerning the nature of the situation. Users may also be biased by some event that they have stored in their long term memory. The user may thus be heavily guided in his/her cue seeking based on some hypothesis that may have already been tentatively chosen. This results in a bias to seek those cues that confirm the pre-determined, but possibly false, hypothesis[1]. Therefore, one of the greatest challenges faced by a commander assessing a situation in a battlespace (whether real or virtual) is determining where to focus attention. This research is intended to help the user to counteract these natural tendencies through the use of virtual environment based training

[1]The best way to test whether a hypothesis is true is to determine whether characteristics, or symptoms, exist that prove it false ([Was72]).

and commander situational awareness aids that are applicable to both virtual and real-world battlespaces.

Historically, commanders have been prepared to face the fog of war using field exercises and board-simulations. However, these techniques do not accurately portray the diversity or complexity of the battlefield environment. In recognition of this problem and to address this concern, ARPA sponsored the SIMNET distributed virtual environment project ([Tho88]). The environment distribution approach, portrayed in Figure 2.1, uses several networked virtual environment stations (using long-haul and/or local connections) to form a single environment wherein each node has its own local model of the environment and there are no clients or servers (see [Bes92], [Bla92], [Bla93], [Fal93], [Pra92], [Tho88], and [Zyd92]). Each distributed simulation host node broadcasts the significant changes in the host's state to all the other nodes, thereby allowing the participants to interact at a distance and to maintain a local model of the distributed virtual environment that is accurate. Each distributed simulation participant, or host, has the same terrain description[1], the same geometric description for the actors[2] in the simulation, the dead reckoning model used by each of the other actors, and identification for the actors involved in the simulation. To accurately maintain the state of the simulation, each host knows the velocity and position of the other actors.

Anecdotal evidence from this ARPA project suggests that virtual environment training effectively prepares individuals and small groups for the complexity and confusion of the battlefield. This evidence is supported by studies of pilots and air traffic controllers that prove that training in a realistic simulation environment transfers to the operational environment ([Car73]) quickly and inexpensively. The requirement that the simulators offer realism is vital to their successes, and is based on Thorndike's common elements

[1]Or as close as possible depending upon the rendering capabilities of the simulators. Standardization and interoperability of terrain descriptions remains an open research issue.

[2]Also called players in some of the literature.

theory ([Tho31]) that suggests that transfer occurs to the extent that a simulator and the environment simulated share common elements.

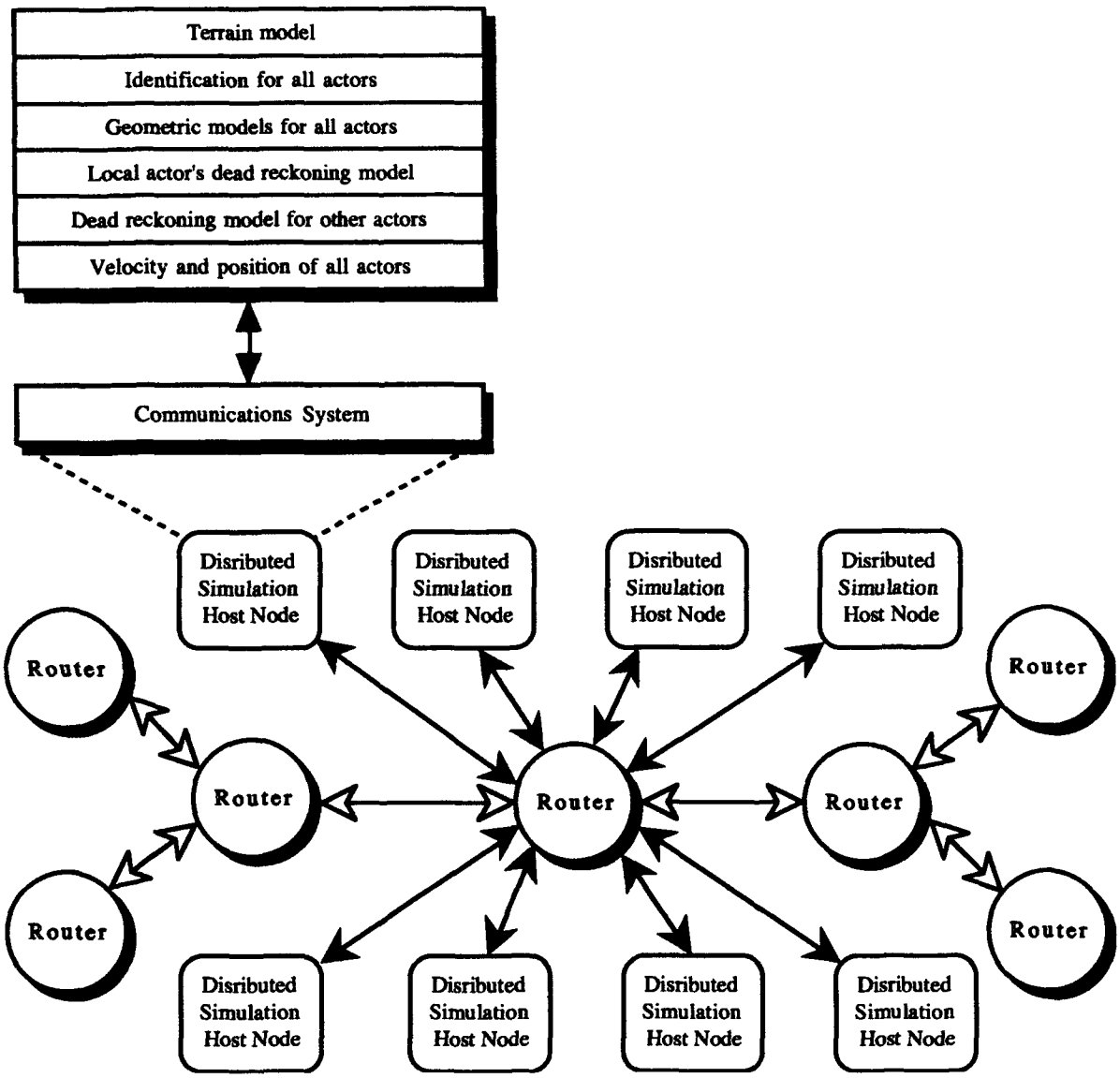


Figure 2.1: The Environment Distribution Approach to the Implementation of a Distributed Synthetic Environment.

2.2 Situational Awareness Tools

Little work has been done to date to develop tools that help higher level commanders comprehend the large stream of data fed to them about a battlespace. The information presented to a commander during a battle should be as easy to assimilate as the information presented to a fighter pilot during air-to-air combat. The information should be clear, concise, readily usable, and directly to the point of winning the engagement. Irrelevant information should be excluded and low level information should be coalesced into higher level information. One way of viewing the situation is that the unit is to the commander as the airplane is to the pilot. Therefore, it would be appropriate to provide a set of instruments that inform the commander about his unit, much like the pilot's gauges tell him the status of the aircraft. The commander should have devices to assist in identifying and resolving problems occurring on the battlefront for the unit. For example, one "fault" might be that enemy forces are approaching the unit and the commander needs to assess the danger to the unit and the impact on his mission of the presence of the approaching force. This thesis effort develops tools to help the commander assess and respond to real-world warfighting situations as well as gain and maintain situational awareness.

Plan-view displays and high-altitude observation posts that view the synthetic environment have been used as one means of improving the commander's situational awareness. Typically, these displays either obscure detail in the battlespace or present the entire contents in one view. The first approach runs the real risk of hiding important information from the commander. The second provides little help since the commander must still sort through all the information. In my opinion, these techniques do not adequately assist a commander in forming and maintaining a mental model of the battlespace. I concluded that users need help determining where to focus their attention and assessing the importance of information outside their field of view but should be aware of.

To reduce battlespace informational complexity, and also improve the commander's situational awareness, one of the goals for this thesis project is to allow the commander to monitor, in real-time, interesting activities within the battlespace without exceeding the commander's capacity to process the data.

2.3 Synthetic Virtual Environments

A survey of all the systems that are currently using virtual environment technology is beyond the scope of this thesis. The review is limited to previous work that is particularly relevant to this research. The work at the University of North Carolina at Chapel Hill (UNC), ([Air90], [Ber93], [Bro86], [Bro88], [Chu89], [Mos86]), is relevant because of its aim of improving the understanding of complex, spatial data using virtual environment technology. There are several interesting similarities even though their application areas are architectural design, molecular modeling, and radiation treatment planning and the research reported in this work supports battlespace visualization, understanding, and analysis.

Comprehending the complexity and interplay between elements of large-scale realistic plans, whether for a building or a military action, is beyond the ability of most people, let alone perform an in-depth analysis for potential conflicts. Architects have addressed this problem using 2D blueprints to convey plans to builders and clients. However, this form of presentation still places a significant cognitive load upon the viewer, and generally leaves the non-architect without a grasp of the spatial relationships in the structure. Radiation treatment planning is a delicate procedure, requiring the doctor to focus high voltage radiation beams upon a patient so that a tumor is killed without destroying surrounding healthy tissue. Because of the spatial complexity of the task and its health risk, radiation treatment planning is performed using templates (protocols) that the doctor and technicians modify to suit each patient's situation. The UNC group posits that it

might be advantageous to be able to generate treatment plans for each patient by interacting with a portrayal of the patient's anatomy (as depicted using 3D medical imaging in a virtual environment) to place the treatment beams in space. This type of interaction may allow the physician to construct a better treatment plan because some of the uncertainty of beam interactions is eliminated through visualization and interaction. Finally, in molecular modeling the location of active sites, the bending of bonds, and the interaction of molecules are extremely complex spatial tasks. Here, the UNC research seeks to enhance understanding by allowing a researcher to view the molecule(s) in a virtual environment and sense the strength of interactions with a force-feedback device. In general, the UNC approach seeks to present sensory events so that they are processed within the context of the user's knowledge of the real 3D world, thereby using previous 3D experiences to give meaning to sensory experiences within the synthetic environment. Their expectation is that a virtual environment system can provide a valuable adjunct to the 2D presentation of complex 3D data.

Other virtual environments have been described in the literature. These serve a variety of purposes, such as immersing an observer in an environment to observe the geometry of curved space-time ([Bry92b]), CAD ([Wei89]), telerobotics and virtual workstations ([Fis86a], [Fis86b]), examining n-dimensional virtual worlds ([Fei90a], [Fei90b]), drawing in virtual space ([Sch82]), and performing surface modeling and virtual environment construction from within a virtual world ([But92]). Environments to enable a person to enter and use a virtual laboratory to conduct experiments ([Mer90]), to perform aircraft mission planning ([Zel92]), to conduct cooperative group work using distributed synthetic environments ([Fah93]), to view and interact with atomic-scale data acquired with a scanning-tunneling microscope ([Tay93]), and to analyze complex economic and business data ([Smi91]) have also been described in the literature. Pausch's work ([Pau91]) describes the design of a low-cost virtual environment interaction testbed built

around inexpensive components for the display device, computing engine, user motion tracking, and gesture input. Like the more recent of these projects, the goal is to allow the user to vicariously experience a virtual world that is outside the everyday experience of humans. The effect of the experience is heightened by immersing the user within an environment that can not be experienced in the real world because the environment portrayed is several orders of magnitude larger or smaller than the human user. In this implementation, the virtual environment is several orders of magnitude larger than the human user.

2.3.1 Computation Distribution Approach

The virtual environments that have investigated both the distribution of computations and the dispersion of the virtual environment among multiple hosts are related to the thesis work currently being conducted. The computation distribution approach, see Figure 2.2, typically partitions the workload among several cooperating machines using a single shared model of the virtual environment (as in [App92a], [App92b], [Bla90], [Cod92], and [Hil92]). Figure 2.2 portrays some of the computations required to realize a synthetic environment that can be distributed among multiple processors. In some cases, the computations indicated by one CPU bubble could, and should, be accomplished by several CPUs to minimize the throughput bottleneck caused by some computations, such as rendering. The model need not necessarily reside in a single shared memory but may be divided among several machines, wherein each machine only possesses that portion of the model that is relevant to its tasks in effectuating the virtual environment. The cooperating machines typically use message passing to update the portions of the distributed model residing in other machines.

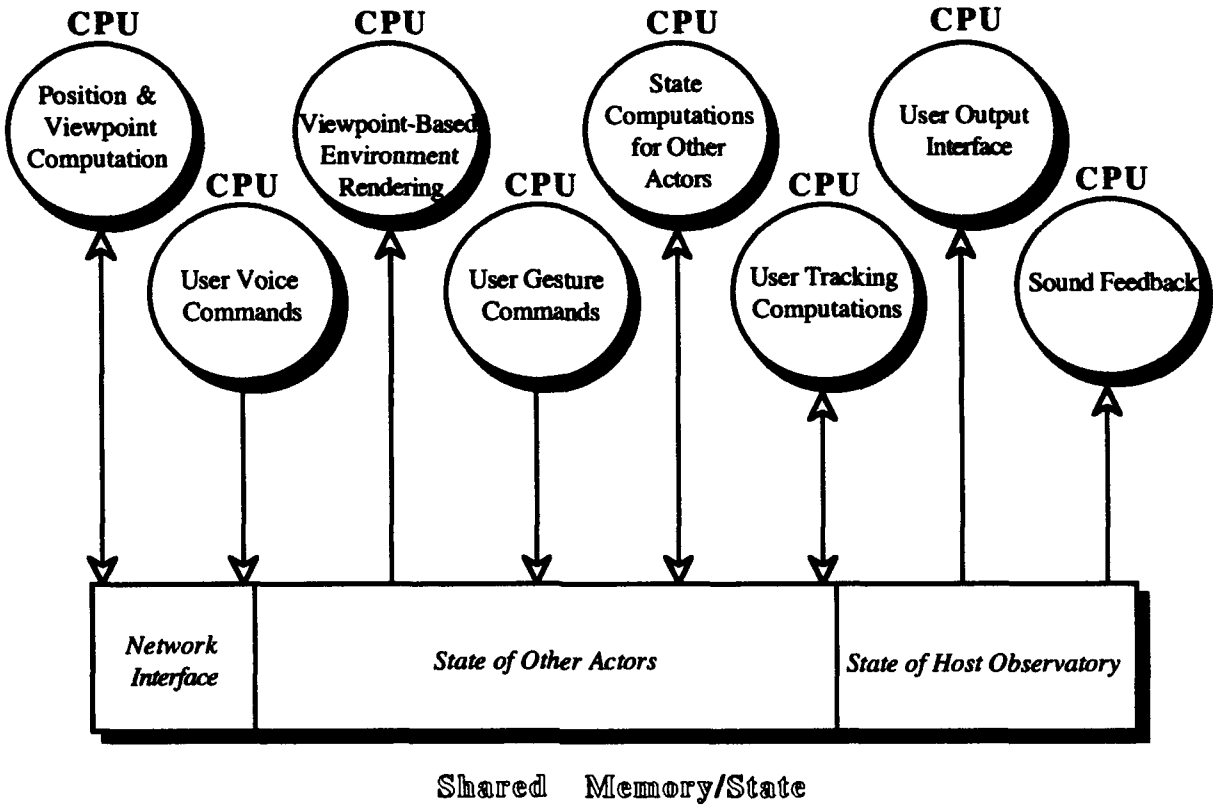


Figure 2.2: The Computation Distribution Approach to the Implementation of a Synthetic Environment.

In the distributed computation implementation, consistency of the environment between CPUs is not a major concern since all the machines share a common description of the environment and the environment is readily updated. However, computational bottlenecks may arise out of the need to update the model's shared description in memory using either a low-bandwidth network or the computer's bus before computing the model's new state and rendering the environment. The distributed environment implementation must solve the environment consistency problem, which can only be partially addressed by dead-reckoning since visual display consistency is also a problem (see [Fer92]). The use of multiple CPUs to perform communication, audio, user interface, consistency computations, and rendering is commonly used within distributed simulation environments

to maintain an acceptable frame-rate and to present the user with a reasonably accurate portrayal of the virtual environment. In this regard, the distributed environment systems build upon the work of the distributed computation systems and contribute to the realization of distributed interactive simulations.

Distributed interactive simulation uses heterogeneous hosts using a common synthetic environment definition to insert a wide variety of both human and computer controlled actors into a single, shared synthetic environment. The hosts are connected using high-speed (currently T1) data links and use a common simulation and network protocol to communicate. The protocols currently in use are DIS and SIMNET (see [BBN92], [Bla93], [Har91], [McD90], [McD91], [Mil88], and IEEE standard 1278-1993). Each host maintains a description of the virtual environment, the actors in the environment, and the motion of the actors in the environment. To reduce network traffic to manageable levels, each actor informs all the hosts of the appropriate dead-reckoning algorithm to use to predict its motion between broadcasts. Each actor runs its own dead-reckoning algorithm and broadcasts new position and velocity information whenever the position predicted by the algorithm significantly differs from the actual position or at the end of a time-out period.

Figure 2.3 contains a conceptual diagram of the overall set of systems used to implement a distributed synthetic environment. The initial processing takes an input terrain description (typically Defense Mapping Agency digital terrain and elevation data) and converts it into a polygonal description (possibly multi-resolution, see [Fal93]) of the terrain to describe the static elements of the synthetic environment.

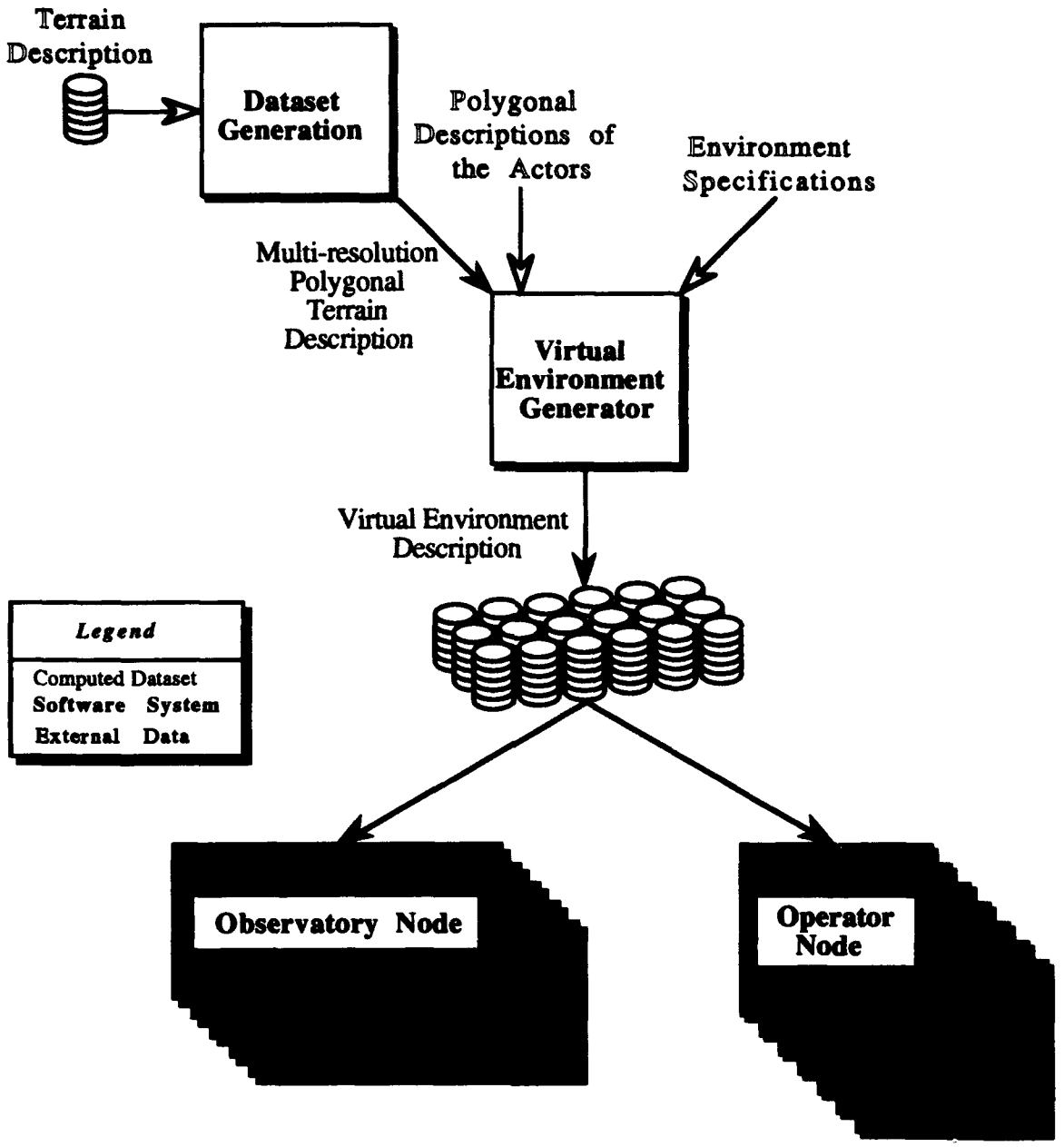


Figure 2.3: Generalized System for Constructing a Virtual Environment.

The Virtual Environment Generator combines the terrain description with the environment specifications and polygonal descriptions of all the actors in the environment to complete a description of the synthetic environment. The complete description contains multi-resolution descriptions of the terrain, man-made structures, and the actors in the

synthetic environment. The Virtual Environment Generator output files can be used for rehearsal, planning, and training of operators and observers of the virtual environment. Within the observer component, the user enters a virtual world that displays the synthetic environment and the movement of the actors in the environment. The operator component also provides the user with a display of the synthetic environment and the movement of actors in the environment, but it constrains the operator to the types of motion permitted by the user's host actor in the virtual environment. Operator components can be either human-controlled, computer controlled, or human-controlled with computer assistance.

After the virtual environment is constructed, it becomes an integral part of the distributed simulation environment. However, because the nodes on the network use different types of simulators to instantiate their actors there are differences in terrain descriptions and accuracy of motion depiction that occur because of the differences in processing speed and display fidelity. Resolving these problems in a distributed simulation is a current research problem. Using the network, individual operators and computerized simulations of actors in the synthetic environment can interact. Furthermore, computer-controlled and human-controlled operator components can be fed data from previous simulations (actual or simulated) that used the same (or similar) synthetic environment. The results of the individual operator interactions with each other and with the computerized simulations can also be used to update the virtual environment database and to maintain a log for later reference.

Clearly, one way to monitor the action in a distributed synthetic environment is to use an actor. However, a better technique is to use an observatory. Unlike an actor, which is required to broadcast information about its state, an observatory is a receive only node. Its presence within a distributed synthetic environment does not increase the network bandwidth requirement nor does it interfere with the activity at any location. An observatory allows a user to position him/herself anywhere, at any time, in the environment

and watch the activity in that area. The user of an observer component is unconstrained in the type of motion that can be performed. Most immersive observatories are limited to allowing a user to jump from actor location to actor location and "see" the action from the actor's point of view. Immersive observatories can also be used to gain a wide-field perspective on the activity in the synthetic environment.

2.3.2 The Synthetic BattleBridge

The work in reducing battlespace information complexity began with the development of the Synthetic BattleBridge (SBB) ([Had93]). The point of departure for developing the SBB was the realization that implementation of a virtual environment requires the seamless melding of several different technologies. Users of the environment must be given visual and audio cues that are sufficiently accurate to entice the user to suspend disbelief and accept the synthetic environment illusion as being real. In addition, sensors to determine the user's position and orientation and a mapping for them from the real to synthetic world are needed. Finally, devices that allow the user to control appropriate portions of the environment, his/her actor in the environment, and the display of the environment are needed. With this in mind, they chose to implement a synthetic environment using commercially available technology and object-oriented design and programming in order to maintain flexibility in the implementation and in the devices that we can attach to it for user input as well as visual and aural feedback. This choice was appropriate since it has allowed us to refine the Synthetic BattleBridge design over time to its present state. The Synthetic BattleBridge is an immersive observatory for large spaces, much as the Virtual Windtunnel is an observatory for aerodynamics effects around an airframe, ([Bry91], [Bry92a] and [Lev92]), and the Virtual Planetary Explorer is an observatory for satellite data about planetary surfaces ([Hit92], [Pic92]). The same concept has also been used for interactive walkthroughs of architectural models ([Air90], [Fun92],

[Tel91]) and analysis of molecular docking ([Ber93], [Bro90], [Min88]). The SBB project was begun in recognition of the fact that the situational displays then in use placed a large demand on the cognitive processes of their users. Part of this shortfall stems from the fact that three-dimensional (3D) data is presented in two-dimensions, thereby forcing users to mentally construct a 3D model of the action within a space. This mental transformation can be error prone, particularly during times of stress. The Synthetic BattleBridge project's goal is to develop a system that allows users to make decisions in an accurate and timely manner by providing several different types of cognitive support for performing analysis. The Synthetic BattleBridge will eventually allow users to evaluate and interact with large-scale (up to several hundred thousand cubic miles) synthetic environments as well as to visualize the activity within a real-world battlespace of the same size. The SBB is designed to provide a visually rich environment that is useful as a training and operational system to a wide variety of users, from firefighters, air traffic controllers and orbital analysts to a combat theater commander.

The SBB is a platform for developing and evaluating advanced user interfaces, information aggregation techniques, and information presentation techniques for presenting synthetic environment generated data in a clear, concise, and accurate manner. The SBB is also a platform for devising and investigating techniques that facilitate information manipulation and user interaction in a virtual environment. The Synthetic BattleBridge functions as a simulation and training platform that provides a capability for participants to interact in real-time when performing group and individual tasks involving mission visualization, mission planning, and training for commanders and planners. Finally, the SBB is a platform that can be used operationally to help users comprehend and evaluate a real-world battlespace. So, for example, a commander can move to any location in a battlespace, observe the activity there, review the information presented about the area being visualized, and analyze the situation without interfering with the action that is

occurring. Or, as another example, air traffic controllers at major airports throughout the world are faced with an overwhelming amount of data concerning the positioning and status of aircraft and ground vehicles and the availability of runways and navigational systems. Because of the complexity of the environment and the rapid changes that occur in it, the controller is usually in the tenuous position of mentally filtering the data before processing and acting upon it. This filtering process can result in the oversight of critical information with life threatening consequences. A partial solution to these problems is a real-time three-dimensional representation of the control-space that depicts the aircraft and ground vehicles in the environment.

To provide these desired capabilities, the Synthetic BattleBridge immerses a person within a 3D, large-scale, virtual battlespace using local- and wide-area network technology and general-purpose workstations with Polhemus sensors, voice control, audio cueing, and color helmet-mounted displays. By design, the SBB is capable of interacting with distributed simulations taking place on the ARPA Distributed Simulation Internet (DSI). Because we immerse the user, we can capitalize upon the human perceptual system's physiological cues[1] and depth cues provided by the traditional computer graphics techniques[2] to impart a feeling of being within the computer generated environment. These technological capabilities provide the SBB with a wide range of realistic and varied scenarios for evaluating its operation and for training because the commander is faced with a situation that in some ways more closely resembles real-world situations than the field or board exercises of the past.

The goal for the user interface and display is to give the user the impression that the battlespace and each object in it are instrumented to provide the information needed by the user to make decisions. The SBB is intended to function as a perfect assistant, providing

---

[1]The cues commonly cited as being triggered are movement parallax, motion perspective, and binocular parallax (aka. stereo vision).

[2]Such as the use of shading, shadows, hidden-surface removal, and perspective computations to render images.

requested information about the battlespace as it is asked for. The information can be low-level, unanalyzed data or data that has been analyzed by an autonomous agent. The only restriction placed on the available information is that data not available in the real world is not to be provided to the user. This restriction is necessary so that the user does not become dependent upon types of information that are not available within a real-world battlespace. Currently, unanalyzed information is presented to the user about the objects and terrain. Regarding the objects, the SBB can provide information about armament, speed, position, damage, type, alliance, force concentrations, missile (ground and air) launch points, and direction of movement of formations. For the terrain, it is portrayed itself, its roughness/smoothness, and major buildings, roads, railroads, etc. in the synthetic environment.

This thesis describes how the Synthetic BattleBridge's immersion effect has been augmented with a tool for the user of the SBB that provides a capability for remotely monitoring the activity at specific locations in a space and allowing the user to view aggregate information about the space. These capabilities allow the SBB user to determine where activity is occurring in a large, complicated space and to assess its importance. The first capability helps the user to detect critical activity in areas beyond visual range. The second capability provides the user with an aid for analyzing activity over a large area. These characteristics equate to increased situational awareness.

The SBB places the user within the synthetic environment using the components depicted in Figure 2.4. The figure also presents a notional portrayal of the possible interactions across the Distributed Simulation Network. The network interconnectivity allows the observatory to witness and analyze distributed simulations composed of interactions between one (or more) previously recorded simulation sessions (via log tape replay) as well as with both human and computer controlled simulation objects[1]. The user

[1]Computer controlled simulation objects are also called semi-autonomous forces or SAFOR.

support functions allow it to generate displays of desired portions of the environment and provides the interface that the user employs to control the observatory. We have identified eight generic objects (components) for the observatory node. These objects are: Network interface, Local environment database, Observatory position and viewpoint, User interface, Local situation, Models for all actors, Observatory viewpoint-based rendering, and Display drivers.

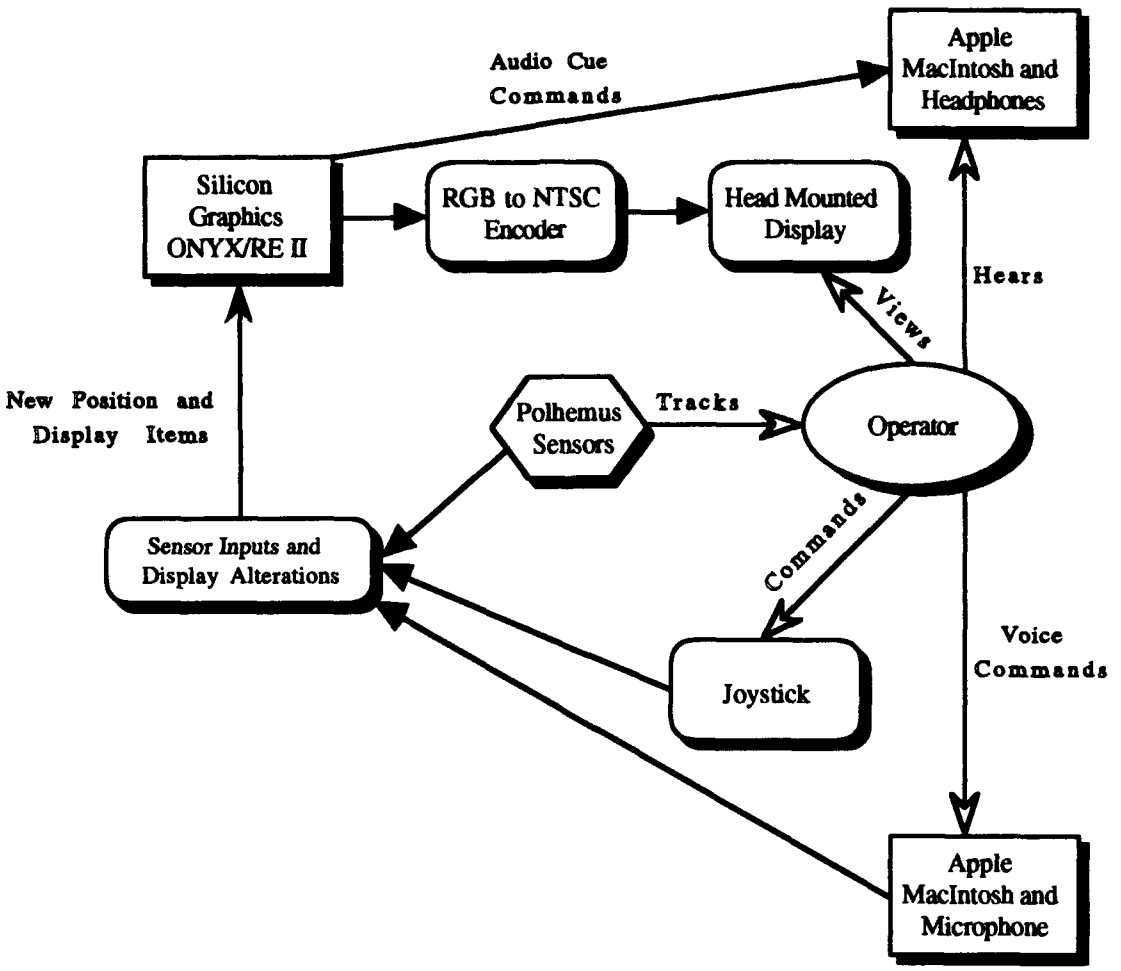


Figure 2.4: Observatory Node Components.

The network interface module implements the network interface using a simulation protocol (DIS, SIMNET, etc., (see [Bla93], [McD90] and [McD91]). The network interactions allow it to maintain an accurate, local copy of the virtual environment. The local environment database module is responsible for updating the database at the node to reflect the database changes broadcast across the network. The observatory position and viewpoint detection module is part of the interface between the synthetic environment and the user. It responds to the user's movement and/or orientation commands and passes them to the display unit for use in rendering the scene. The local situation module maintains the status of the local synthetic environment (the situation within visible and/or sensor range) for use by the rendering module. The local situation database is a subset of the global database maintained by the local environment database update module. The other actors module holds the descriptions of other actors in the battlefield. This information includes, but is not limited to: vehicle exterior description, dead-reckoning algorithms, and sensor capabilities. The observatory viewpoint-based rendering module calculates an image that portrays the synthetic environment from the viewpoint of the operator. This module performs all the image rendering functions that are required for the type of observatory that the node supports. For an immersive observatory, the module performs hidden-surface removal, texturing, shading, stereo display (if desired), and shadowing. For a "true 3D" observatory, as in [Hob93], the type of rendering computations are determined by the "true 3D" display device.

The major SBB components and their interactions are presented in Figure 2.5. The component choices have been made with the intention of providing the user with a direct manipulation hands-free interface. A Macintosh computer, with the Voice Navigator voice recognition system and a wireless microphone is connected to the Silicon Graphics computer to provide the user with hands-free control of the system. Several display technologies are available for synthetic environment display: a locally built color LCD head

mounted display (HMD), the Polhemus Looking Glass™ fiber optics color CRT-based HMD, and the Fake Space Labs BOOM2M™ monochrome CRT-based system. Viewer position and orientation when using the HMDs is obtained using a Polhemus 3-Space Tracker™ magnetic sensor attached to the HMD. The BOOM uses an internal mechanically linked tracking system to determine viewer position and orientation. Viewer movement through the display volume is assisted by a hand-held two button mouse for the HMDs or the interaction buttons on the BOOM.

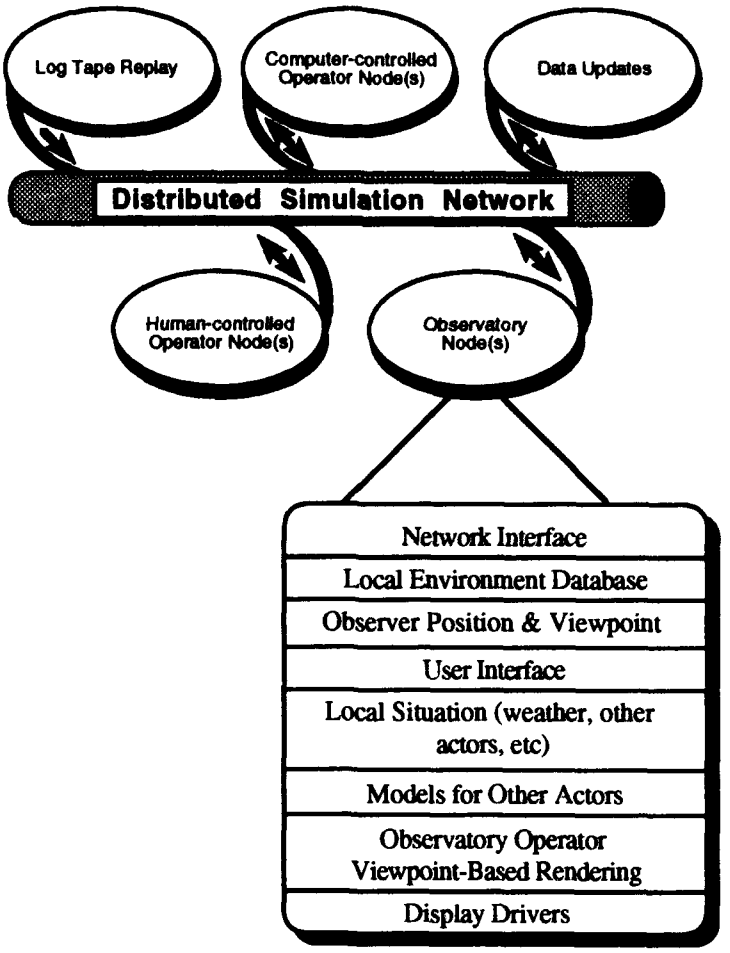


Figure 2.5: Synthetic BattleBridge Architecture Schematic.

This combination of movement techniques reduces the amount of physical movement required by the user and helps ameliorate the limited range problem encountered when using magnetic tracking technology. The user's movement is unrestricted, the user is allowed to move to any location in the virtual battlespace at any velocity without physical or material restriction on the movement. There is also a terrain following feature, this allows the user to move along the terrain at a low altitude at high speed. This type of movement is sometimes useful to users when moving between locations.

The SBB is an immersive virtual environment observatory. The SBB is designed to allow users to monitor, analyze and evaluate large-scale (several hundred thousand cubic miles) virtual environments. The initial goal for the SBB was to give a user a sense of the spatial orientation, type, motion, and distribution of objects in a synthetic environment. Key issues included the ability to display real-time data at interactive display rates and to provide a very large scale, immersive environment with a large range of object types, sizes, and speeds. The SBB provides these capabilities by computing vehicle position, motion and velocity data and presenting this information, in real-time, using a three-dimensional rendering of the battlespace. The raw, unanalyzed information is presented using a combination of visual icons and text. For example, threat envelopes are displayed for active surface-to-air missile systems (SAMS) and anti-aircraft-artillery (AAA) vehicles. Envelopes are derived from unclassified, published data and assume maximum capabilities without consideration of terrain or atmospherics. Radar envelopes are displayed for active, emanating SAMS, AAA, and radar systems. Radar envelope display criteria is the same as that for threat envelopes ([Had93]). Locators, which are semi-transparent bubbles, are placed around objects to help the user to locate various objects in the environment. The locators can be activated selectively for different types, or classes, of objects. Aircraft trails are displayed for all active aircraft and missiles. Trails show the flight path of the vehicle over the previous fifteen seconds. Missile tracks are displayed for all active and de-active

missiles ([Had93]). Tracks show the entire trajectory of the missile from initial activation to deactivation or impact. The tracks remain viewable for the entire user session. The user is also allowed to interactively designate up to one hundred locations in the battlespace as viewpoints (both a position and view direction are specified) at any time during the distributed simulation session and can move to any of them at any time using a voice command. The system also allows users to attach to any vehicle and move with it as though physically tethered to it. A plan-view display option is also provided, the user can call up this form of display at any time.

2.3.3 Other Virtual Battlespace Environments

Others have also built virtual environments to allow operators to visualize battlespaces as well as interact and orient themselves within battlespaces (see [Bes92], [Bla92], [Fal93], [Pra92], [Pra93], [Tho88], and [Zyd92]). The work reported by Falby, Pratt, and Zyda ([Fal93], [Pra92], [Pra93], [Zyd92]) on NPSNET complements our own, especially in regard to their work in implementing large-scale virtual environment battlespaces on commercial workstations. They implemented a distributed system that allows users to view the activities of multiple actors within a medium-scale virtual battlespace as well as place actors into the environment. They provide a 2D plan-view to allow users to orient themselves within an environment and auditory cues to enhance the sense of realism provided by the visualization of the environment. They do not provide other assistance to help the commander determine where to direct his/her attention or to help the commander to assess a situation.

2.4 Fuzzy Logic Controllers and Their Uses

The Fuzzy Logic Controller is intended to mimic, as much as possible, the way human beings actually think and interact with their environment. The basic concept stems

from the fact that humans do not think in "crisp" terms. When humans evaluate something, they do not give that evaluation a single value, but rather they assign a range of values to that concept. At the boarders of this range they make decisions as to whether something belongs or does not belong in uncertain or "fuzzy" terms. For example, what do we actually mean when we say a person is tall? Where is the start point and end point for the concept of "tall"? In other words, humans think in terms of degrees of membership which relates directly to the way they perceive the world around themselves. Fuzzy Logic Controllers try to duplicate this thought process by developing membership functions and rules that can be associated with the concepts that we are trying to model.

The uses for Fuzzy Logic Controllers stem from the need to model control problems for which no mathematical model exists nor can the model be developed in a reasonable time ([Alt92]).

> The advent of fuzzy control systems has dramatically transformed the control problem from one of exact mathematics, to the encoding of inexact, commonsensical inference rules. This approach, besides being intuitive, has the rewards of flexibility, ease of implementation, and elegance. Furthermore, an increasing number of complex processes that could not be previously automated are now machine controlled by fuzzy control systems ([Tob92]).

2.4.1 Key Definitions and Concepts

The following list contains key definitions and concepts needed to fully understand the following paragraphs:

- **Fuzzy Set:** a class of objects with a continuum of grades of membership.
- **Membership Function $f_A(x)$:** associates with each object a real number in the interval [0, 1], with the value of $f_A(x)$ at x representing the grade of

membership of x in A. Thus, the nearer the value of $f_A(x)$ to unity, the higher the grade of membership of x in A.

- **Linguistic Terms:** words which represent a quality that can be ordered into a natural hierarchy (large, medium, small, etc.).

2.4.1 Linguistic Evaluations in Risk Situations

C cía purposed that the calculation of the global risk of a structured system can be found by evaluation the risk of each individual component ([Gar92]). This can be accomplished in three steps. The translation of natural language expression to fuzzy set notation is the first step. The next step is to combine all the fuzzy sets into a single weighted value which in itself is a fuzzy set. The final step is to take the results from the previous step and match that fuzzy value to the nearest natural expression that was introduced in the first step. This final result is the risk of the entire system based on the individual components of the system ([Gar92]).

The key to this process is in determining the proper set of natural expression to be used. If the set is too restrictive or partitions the set space into broad categories then a good deal of fuzziness is lost. We can therefore uses hedges to further partition up the natural expressions. For example, instead of just using "big" we could use "very big", "extremely big", or "more or less big". The terms before "big" are the hedges which further break up the definition of the term "big" thereby introducing more fuzziness into the natural language ([Gar92]). These terms are commonly called linguistic variables.

2.4.2 Use of Color for Decision Making

Benson shows how through the use of fuzzy set theory, subjectively defined categories can be presented to an analyst to help support decision making. For display

data, Benson shows how the fuzzy nature of color can represent data in various categories on a graphics terminal for visual inspection. Benson achieves this by giving two examples in which the degree of color visually represents the subjective data presented.

The first example shows how the deliberate blurring of category boundaries can correspond to how a color can gradually transform from one recognizable color to another. In this example the use of color going from yellow to orange to red is used. The blurring of data (color) represents the uncertainty or fuzzy nature of the information. Benson states three reasons for the deliberate blurring of data:

> In general, deliberate blurring is a useful strategy for at least three reasons: undue precision is not needed for the purpose at hand; the data itself is imprecise; and the level of anxiety in decision making is reduced. ([Ben82: 430])

The second example is an extension of the first example. However, in the second example we now combine several subjective variables or categories together to get an overall evaluation of all the data represented. Each variable contains a color that represents the degree of fuzziness of the associated data. Once all the information contains the correct color coding, manipulation of all the fuzzy variables can be performed. By using various fuzzy mathematics, the variables combined represent the overall evaluation of the data. Also, note that by changing the mixture of the variables to represent the importance of some variables over others, we can come up with a different color coding of the overall performance.

Benson presents us with a color coding of membership functions for simple linguistic terms and expressions. Benson then shows how color can represent the different degrees of linguistic terms as applied to the fuzzy set concept. ([Ben82: 432]) For example, if we let red represent the greatest degree of membership and yellow represent the lowest degree of membership (without being zero), then a blurring of color from yellow

through orange to red instantly gives us the information we need about the particular relationship.

Using fuzzy set theory to analyze and display subjective categories of data for decision support and decision making frees the analyst from many cognitive and memory tasks. The analyst can simply view the color coding and understand all the needed relations between the data at hand. The use of color as a visual representation of the fuzzy concept works extremely well. The natural perception of one color slowly blending to another in a constant degree of change relates to the idea of the degree of membership for a fuzzy set going from zero to unity. Combine this with the fact that humans can perceive color information faster and more accurately then looking at raw data, and we get an efficient and useful way to display subjective information. Color allows the viewer to shift easily between two perceptual attitudes: association (disregarding variation in order to see similarities) and selection (distinguishing variation to isolate similar instances). ([Ben82: 436]) This by itself allows the analyst the ability to look at the subjective date from various view points and perspectives.

2.4.3 Use of Weights as Applied to Fuzzy Rules

This subsection deals with the use of weights applied to the rule base that controls the processing of the overall fuzzy value to be associated with the system in question. A number of papers address the use of experts that evaluate the rule base and assign values to each rule. The values represent the importance of the rule to the overall evaluation of the system. A higher value indicates a rule that should be considered more important to the calculation of the finally fuzzy value. These evaluations of the weights are themselves expressed as fuzzy numbers. In this way there is no loss of generality when the final calculations are performed ([Ram92]).

The model proposed by Ramakrishnan, uses the opinions of multiple experts on a small subset of the entire rule base in assessing the weights for the system ([Ram92]). On the other hand, the paper by Qiao uses a four step process to improve the rule base:

(1) translating the operators' experiences into fuzzy linguistic form directly;
(2) monitoring and summarizing the control behavior of the operators;
(3) modeling the process to be controlled, using fuzzy set theory;
(4) self organizing in running of the control systems. ([Qia92])

Both these authors present test results that show the fuzzy control system obtains an optimized performance according to what the designers hoped, and this demonstrates the effectiveness of fuzzy set theory in imitating human thinking ([Qia92], [Ram92]).

2.4.4 Implementation of a Feedback Controller

This unique method for implementing a Fuzzy Logic Controller stems from the idea used in the design of amplifiers in electronic circuits. If the Fuzzy Logic System has both fuzzy inputs and fuzzy outputs, then we should be able to feedback the output information back into the Fuzzy Logic Controller. Then, just like in a amplifier, this new input to the controller helps stabilize the output results. This then gives us a self correcting Fuzzy Logic Controller. This mimics the way human beings perform a task. The human being is constantly making small adjustments to the way they are doing things based on the results they are getting back from their current output ([Ali92]).

2.5 Conclusion

By combining the idea of situational awareness with the fuzzy set theory concept, a system can be developed that could mimic scouts reporting in from the field during a large battle to report troop movements in areas of interest. We can combine the idea of the use of color stated by Benson, to represent the level of activity, threat, or risk in a given area of

the battlespace. Commanders could then glance over to the corner of the screen to see if any of the "hot spots" they defined are active. The degree of color associated with the area, along with its bar length, would give commanders a relative feeling about the current importance of the area. The rest of this thesis describes just how this is possible, along with the design and implementation of the actual Sentinel system.

III. System Design

3.1 Introduction

Complex skills, such as problem solving, are organized hierarchically ([And81], [Car92], [Mic88], [Ras86]). Situational awareness usefully characterizes complex problem solving skills. This thesis effort has been working to determine and incorporate the strategies and sub-goals of the hierarchical skill organization used by commanders to assess a battle situation and maintain situational awareness. A fuzzy logic processed semantic network[1], called the Sentinel[2], within the Synthetic BattleBridge (SBB), captures and processes the resulting problem solving hierarchy. Each of the semantic nodes has a different input to the level of threat in the Sentinel's watchspace at any one time.

Consider the situation of a battlefield commander who must make decisions based upon data gathered using several different modalities. Making these decisions requires that the commander mentally combine the information to produce an overall mental model of a battlespace. There are several reasons for the difficulties encountered in forming the model, determining what the enemy is doing, and reacting appropriately, or even pro-actively. First, the important portions of the battlespace environment differ from moment to moment. At one instant it might be an enemy reconnaissance event, which could indicate the beginning of an enemy aerial operation, a ground engagement, a dogfight, or the arrival of a resupply mission. Second, the commander does not assess in isolation the "interest" value of this and other potentially important enemy and friendly force actions. Instead, the commander judges their values in relation to other events happening at the same time, in the

---

[1]The semantic nodes of the network characterize an object, situation or concept. For example, a situation in an area may have *stabilized*, the enemy lines may have been *penetrated* or *nearly penetrated* , or events along a front may have *settled down* . These nodes can also be characterized as linguistic variables, which is well suited to a fuzzy logic representation, and are all relevant aspects of situation assessment.

[2]The Sentinel contains the fuzzy logic-based situation assessment tools and is designed to monitor the activity within an operator-designated portion of the battlespace. The function of the Sentinel is described later.

same location as well as events happening at the same time at other locations. Third, humans have a time and space limited focus of attention, meaning they are limited by the amount of information perceived and processed at any time. Because the commander is limited in the amount of information that can be perceived and processed at any time, the commander can miss or forget important information about the battlespace. Fourth, the search and discrimination task the commander must accomplish are serial tasks, so processing time increases linearly with the number of objects in the battlespace. Since it takes relatively constant time to process each object, the basic task of locating and discriminating between objects can become easily overwhelming for the commander, let alone assessing their importance. As a result of these four factors, commanders usually lack complete situational awareness of events outside their field of view. Therefore, the development of a software tool, called the Sentinel, addresses these factors and reduces their negative impact on effective decision making. The Sentinel addresses the need for tying together several, disjointed data gathering systems to present a clear, consistent insight into the action within a real battlespace or a distributed simulation-based battlespace.

The Sentinel portion of the SBB helps address the four concerns mentioned above by providing an information consolidation and analysis capability. The commander can place each Sentinel in an area of the battlespace that the commander identifies as a possible location of a future important activity (a watchspace). Figure 3.1 depicts a Sentinel positioned within a notional battlespace. The transparent shaded cylinder volume shows the location of the Sentinel's watchspace within the battlespace. The watchspace assessment bar shows the interest level of activity within each of the watchspaces in the battlespace. The "Island" watchspace, shown in highlighted color in Figure 3.1, has a high level of interest as indicated by the length of the status bar and its color. The other four Sentinels (not pictured) have computed a much lower level of interest for their watchspaces. As activity occurs within each Sentinel's watchspace, the Sentinel

automatically assesses the importance value of the total activity in its space and signals this value to the commander. The commander thereby relieves himself of the necessity of trying to determine the important areas of the battlespace from the raw data, he can make his assessment based upon the ratings provided by the Sentinels.

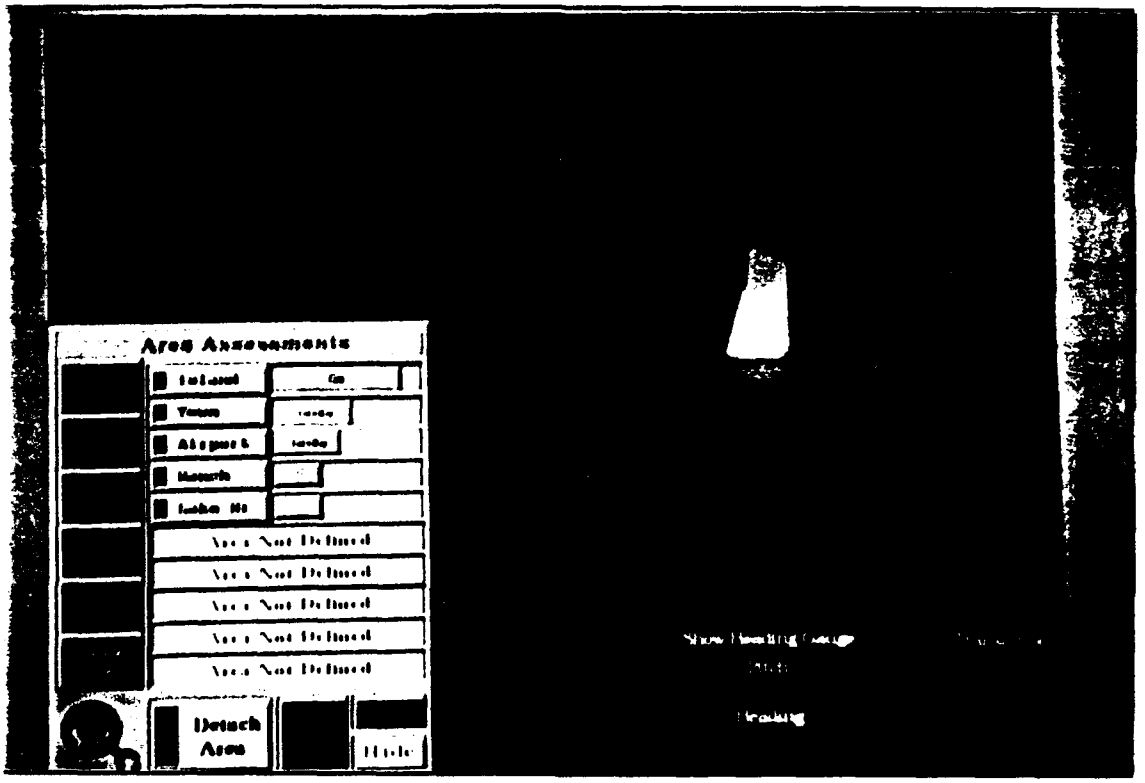


Figure 3.1: Display Showing GO Level of Interrupt for "Island" Sentinel After a Rule Fires, and the Sliding Scales for a Set of Sentinels.

Since each Sentinel can assess the activity of friendly and enemy forces and evaluate these actions, the assessment it provides the commander is a consolidated assessment of the total activity within a watchspace. Therefore, the commander can determine the relative importance of the different portions of the battlespace by simply examining the ratings provided by the Sentinels[1] and reviewing previous Sentinel reports

[1]Of course the commander must insure that all important areas are monitored.

to gain a perspective on the progress of the battle. Note that the Sentinel does not make decisions, it consolidates data and functions as a situational awareness aid for the commander. The Sentinel performs its information aggregation function using fuzzy logic.

The rest of this chapter pertains to the development issues of the design. Section 3.2 discusses the overall design methodology for the entire Sentinel system. Section 3.3 and 3.4 talks about the design of the structured programming units (libraries) and the object-oriented class hierarchy, respectively. Finally, section 3.5 states some conclusions about the overall design of the Sentinel system. Chapter four addresses the implementation of the Sentinel system.

3.2 Design Methodology

The design of the Sentinel takes into account classical structured programming techniques, as well as an object-oriented methodology. This section describes the current design in relation to the above mentioned techniques and methods. This section also addresses the chosen data structures associated with the design of the Sentinel as they apply to design decisions.

A mixture of structured programming techniques and object-oriented methods encompasses the overall design methodology for the Sentinel. The reasons for the mixture of the two technologies stem from the use of an interface tool that does not support object-oriented classes and methods. The next chapter on the implementation of the Sentinel system addresses this interface tool called "Forms Library: A Graphical User Interface Toolkit for Silicon Graphics Workstations." ([Ove92])

The basic design consists of a number of structured programming units (libraries) connected using an object-oriented class structure. This object class then controls how the driving application (in this case the Synthetic BattleBridge (SBB)) implements and uses the Sentinel. By doing this, we have encapsulated non-class procedures and functions inside the class methods. This then only allows access to these non-class procedures and

functions through method calls by the driving program (SBB). While this is not a pure object-oriented design, (the driving application could make calls to the non-class procedures and functions itself), if the driving application applies strict adherence to using only the class method calls provided, then access only occurs through those method calls.

Figure 3.2 shows the overall design of the Sentinel system. The structured programming units show only communication into and out of the Sentinel class structure. The only communication with the driving application (SBB) is through the Sentinel class methods. As long as this applies totally in the driving application (SBB), it appears that the Sentinel has a complete object-oriented design.

3.3 Library Unit Structure

The Sentinel system comprises a number of structured programming units compiled as libraries. These library units make up most of the input, output, control, and fuzzy logic computation, needed by the Sentinel system. The main reason why these library units could not compile as object-oriented classes, stems from the type of user interface developed for the Sentinel system. A non-object-oriented tool (forms 2.1) created this user interface. Due to this, the functionally of the system splits up into five functional areas or units: configuration, input, control, output, and computational.

3.3.1 Configuration Unit

The configuration unit's main task is to set up all the configuration information as needed by the Sentinel system. It does this by reading in a number of configuration and default files that set up the locations of the initial Sentinel watchspaces and the weight factors for all the different types of entities possible in the simulation. The system reads in this information, and enters the information into the appropriate data structures as needed by the other library units in the Sentinel system.

Conceptual View Of Overall System

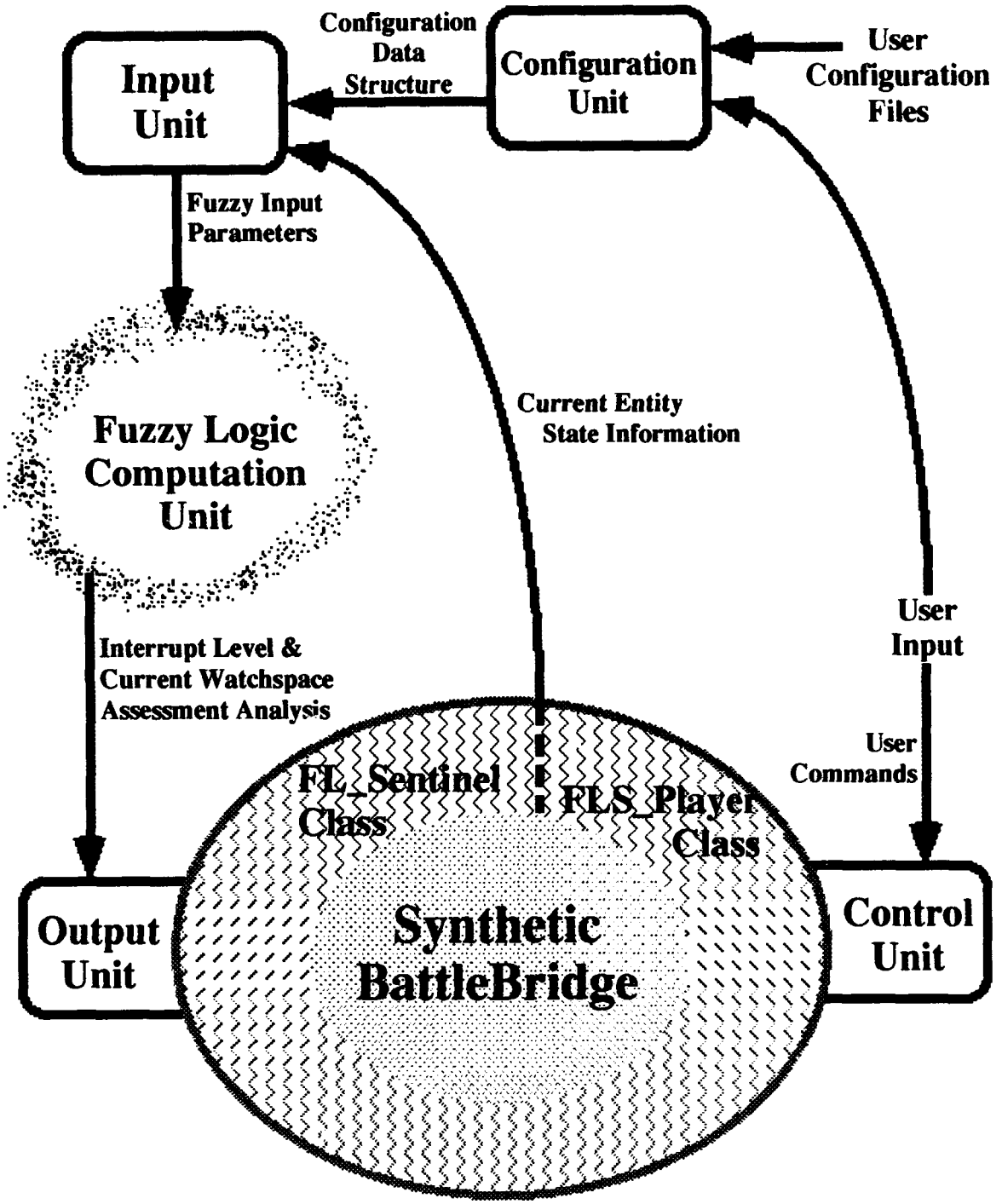


Figure 3.2: System Design for the Overall Sentinel System

The Sentinel system also performs error checking on the configuration files to make sure the files are reasonably correct and consistent. The Sentinel system checks closely the initial placement of Sentinel watchspaces. The positioning of a Sentinel watchspace location in the simulation at start up happens by one of two methods. The first method requires the lat-long position of the center point of the watchspace. The Sentinel system thoroughly checks this lat-long for input errors. The second method requires an x-y-z position of the watchspace. The Sentinel system then places that watchspace in the scene according to a local origin.

The Sentinel system can use both ways with either a flat earth or a round earth representation for the simulation. However, the x-y-z positioning currently uses a configuration file. Changes to this file happen manually with the use of a text editor. The Sentinel system accomplishes lat-long positioning through a stand alone utility that allows the user to set up watchspaces interactively. Other error checking includes making sure that the number of watchspaces specified match the number of watchspaces requested, the number of watchspaces specified does not go over the maximum number of Sentinel watchspaces allowed by the Sentinel system, and the radius for each watchspace falls within the maximum radius size currently allowed for a Sentinel watchspace.

The configuration unit also supplies a tool by which the user can change player entity weights for the fuzzy logic assessment during execution of the simulation. This allows the user to make changes that determine how the computational unit views certain player entities and designations. Chapter four on implementation gives details on how the user can change these weights.

3.3.2 Input Unit

The input unit's main task is to take in current information on the status of the simulation and parse this information into data structures needed by the Sentinel's

computational unit. The input unit contains five functional areas that together perform the above mentioned main purpose.

The first functional area is the initialization phase. This phase, as the name implies, initializes all the data structures needed to process the current state of the simulation. It basically sets the state of the input unit to a state that has no prior knowledge of the current simulation.

The second functional area is the entity checker phase. In large simulations there is the possibility of great numbers of players participating. These players may take on a very diverse number of different types or designations. The Advanced Research Projects Agency (ARPA) Distributed Interactive Simulation project ([Tho88]) currently uses DIS protocols that can broadcast more than 1000 different types of player entities and designations. Therefore, the second phase is a first cut at limiting the number of different entities and designations for the Sentinel computational unit cares about. The entity checker simply goes through a series of switch statements to see if the Sentinel system needs the current selected player entity. As shown in Figure 3.3, this visually represented a tree traversal through all the different types of DIS entities and designations. If the Sentinel system does not need the entity, the Sentinel system ignores the entity and the Sentinel system checks the next player on the list.

The third functional area is the contained-in phase. Once the player has passed through the second phase, the Sentinel system checks the player's position against each Sentinel watchspace to see if the player currently falls within the bounds of that particular Sentinel watchspace. There is one of two containment checks performed on the player location depending on whether the driving application is in flat earth or round earth representation. The Sentinel system bases both calculations on a cylinder representation of a Sentinel watchspace (see subsection 4.2.7).

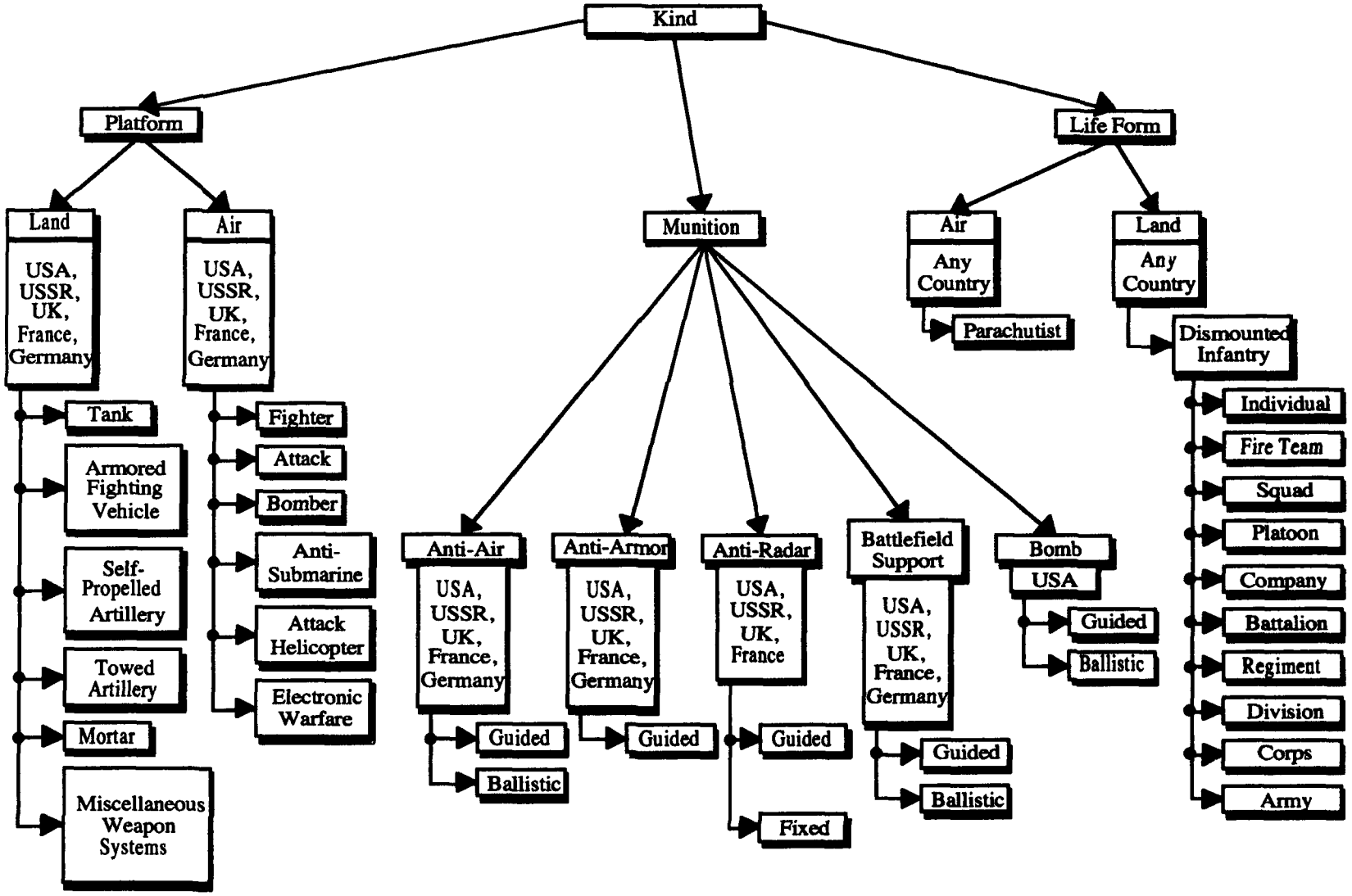


Figure 3.3 Entity and Designation Checker for the DIS Standard

The flat earth calculation simply projects the player onto the xy plane and checks the player's distance to the center of the Sentinel watchspace and compares it to the Sentinel's watchspace radius (see Figure 3.4).

The round earth calculation uses three points to determine the shortest distance from the player to a line extending from the center of the earth through the center of the Sentinel watchspace and comparing this distance to the radius of the Sentinel watchspace. The three points used in the calculation are the center of the earth (0, 0, 0), the center of the Sentinel watchspace (S_X, S_Y, S_Z), and the current player position (P_X, P_Y, P_Z). The Sentinel system applies the law of cosines to find the angle between the player and the Sentinel watchspace with the center of the earth being the common point. Once we know the angle, the shortest distance between the player and the Sentinel watchspace axis is the perpendicular distance between them. We then use the sin function to find this perpendicular distance. Figure 3.5 shows pictorially this calculation.

The fourth functional area is the count phase. Once a player has passed through both the second and third phases, the Sentinel system considers this player an entity that the computational unit needs to know about. The Sentinel system does this by adding the player's entity weight to a count array for the particular Sentinel watchspace that the player is in. This count array has a cell for each of the different entity types and designations denoted as being needed for the Sentinel computational unit.

The final functional area is the processing phase. The processing phase takes place after all the players have gone through phases two through four above. At this time, each Sentinel watchspace has a count array indicating how many of all the different entity types and designations fall within the Sentinel watchspace's boundaries. The Sentinel system now processes these counts according to the information the Sentinel system needs for the Sentinel's computational unit. This "process" can vary depending on how the Sentinel computation unit needs the information and in what form.

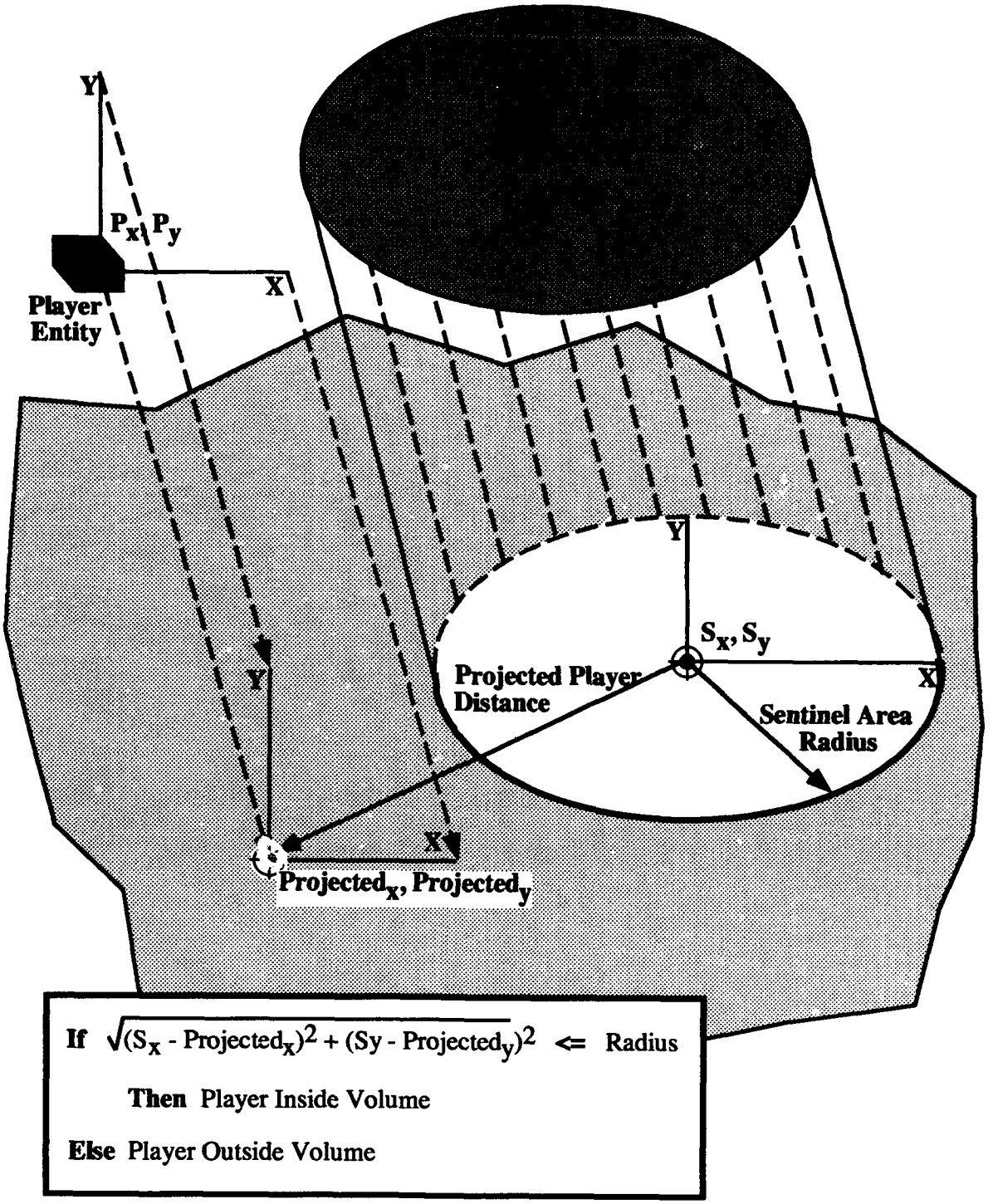


Figure 3.4: Flat Earth Containment Calculation With Cylinder.

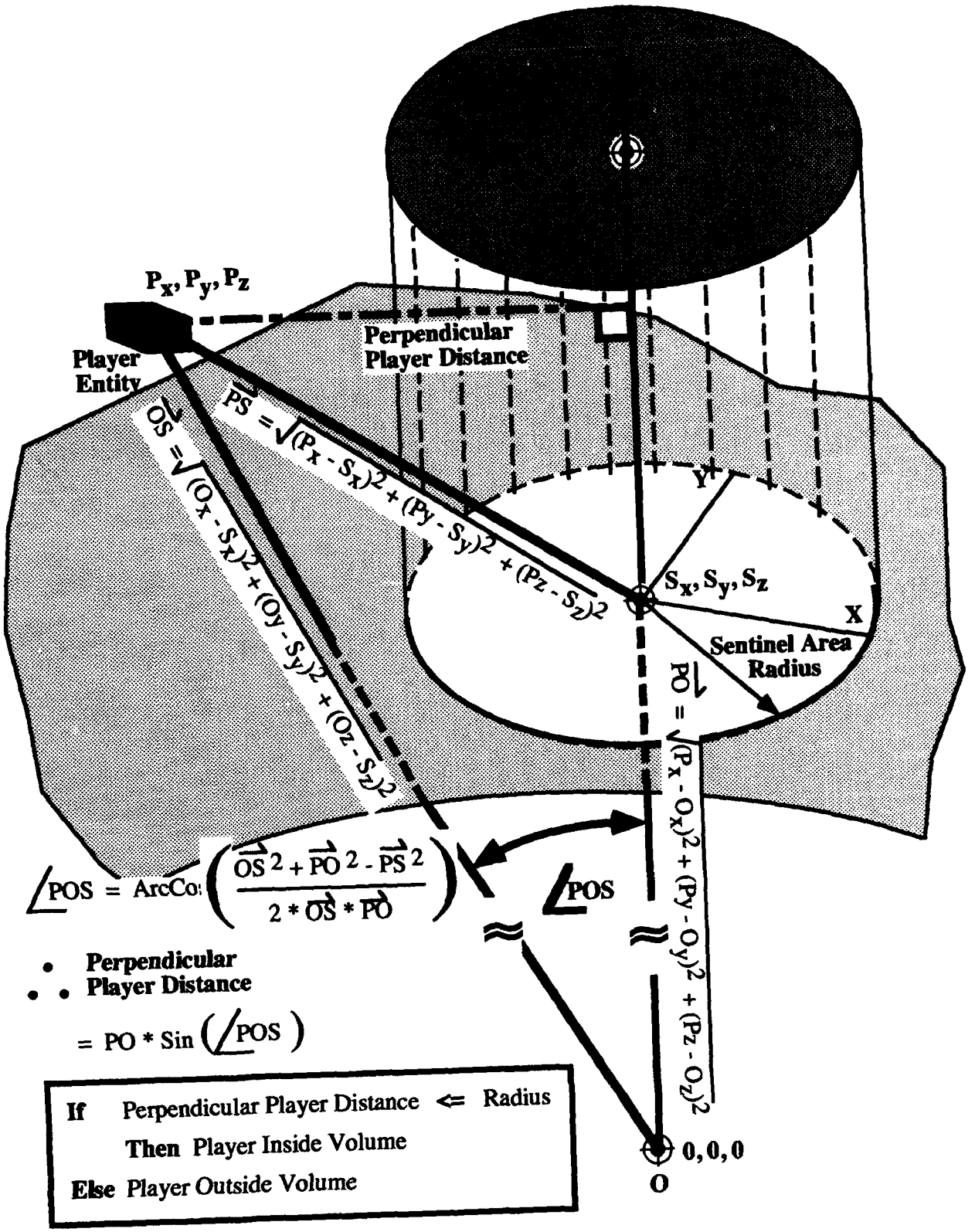


Figure 3.5: Round Earth Containment Calculation With Cylinder.

The implementation subsection 4.2.5 on the computation unit discusses how the Sentinel system performs this function for the SBB application.

3.3.3 Control Unit

The control unit's main task is to interpret user commands through the various control panel interfaces supplied by the Sentinel system. There are two main types of control: Sentinel watchspace manipulation and Sentinel watchspace information visualization. The first type deals with how the user can control the actual Sentinel watchspaces themselves, while the second type deals with how the user can visualize information about the Sentinel watchspaces.

The user can currently control seven different operations that pertain to manipulation of the Sentinel. The following paragraphs discuss the methodology of each operation, while chapter four addresses the implementation.

The first operation is panel control. Panel control simply allows the user the ability to navigate through the different Sentinel control screens that are available. It allows the user to toggle from one mode to another. The rest of the Sentinel watchspace manipulation operations deal with directly modifying or viewing the Sentinel watchspace of interest.

The second operation allows the user to add a new Sentinel watchspace to the simulation provided there is still a slot available for a new watchspace. The current Sentinel system allows for ten defined Sentinel watchspaces within a simulation. The Sentinel system initializes the new watchspaces with a default radius of one mile that the user can modify later as explained later on in this subsection.

The third operation allows the user to attach or detach to or from a given Sentinel watchspace. Attachment automatically puts the user in a view directly over the Sentinel watchspace and high enough above so that the user sees all the Sentinel watchspace's base on the screen. Once there, the user can move about the view as they would with any other

view and set a preferred view point and direction for that Sentinel watchspace. Once done, if the user goes back to that Sentinel watchspace again, they return to the view specified on the previous attachment. The user can also detach from a Sentinel watchspace that takes the user back to the previous view they were looking at before they attached to any Sentinel watchspace.

The Sentinel system can only perform the remainder of its operations if the user attaches to the Sentinel watchspace of interest. Once attached, the user can move the watchspaces, modify the watchspace's radius, reset the watchspace's view, and finally delete the watchspace itself. With the move and modify radius operations, the user can apply a number of changes to the Sentinel watchspace representation. However, the user could reset these modifications to the state they were in before the modifications took place, by pressing a reset command located within the respective sub-control panel. Once the user has accepted the changes with the "ok" command, the Sentinel system permanently alters the Sentinel watchspace to reflect the new position and/or radius change. The last two operations, reset view and delete watchspace, are final and the user can not undo them. The reset view operation simply resets the particular Sentinel watchspace's attach view to the initial view as described above in the attachment operation. The delete watchspace operation removes the selected Sentinel watchspace from the simulation. The Sentinel system declares the watchspace invalid and performs no more computations for that watchspace. However, once deleted, the watchspace's slot then becomes available for reuse by the add watchspace operation.

There are currently two operations that the user can control that displays visual information about selected Sentinel watchspaces. The first operation allows the user to view a strip chart about the history of the Sentinel watchspace as it pertains to past watchspace assessments. The second operation presents the user with a conceptual, two dimensional view of the capability of the player entities within the chosen Sentinel

watchspace. The idea here is to give the user a graphical view of how the player's entity capabilities combined within a given watchspace of interest. The color coding and size variation based on capability and force type (friendly, opposing) gives the user a visual view of where the action is and where the concentrations of forces fall within the Sentinel watchspace. Chapter four discusses the implementation of the above operations.

3.3.4 Output Unit

The output unit's main task is to translate current simulation and Sentinel data into visual information presented on the screen to the user. The output unit has four areas of functionally. The following paragraphs address the design of this unit in terms of its areas of functionally: initialization, display and update of watchspace assessments, interrupt control, and virtual keyboard control.

The first area, initialization, sets up the required global variables and data structures associated with the output unit's functionally. It also initializes the various user interfaces associated with the Sentinel for display and control.

The second area, display and update of watchspace assessments, is the main display method of the visual watchspace assessments made by the Sentinel. It is here that numeric information from the fuzzy logic computational unit is transformed into visual information and then displayed to the user on the screen. The numeric information is quantized into red, green, and blue components used by the display form to produce the desired color based on the given watchspace assessment value. The update function simply performs the display function mentioned above at periodic time intervals. The display and update function also handles a number of maintenance items associated with the display forms. They keep track of what display form representation is currently being view by the user and updates that form appropriately. If the Sentinel system adds or deletes Sentinel watchspaces, the Sentinel system modifies the necessary forms to indicated the changes to

the Sentinel watchspace slots. If Sentinel watchspace names change, the Sentinel system keeps track of this and makes changes as needed.

We initially intended the fourth area, interrupt control, to be a separate unit. However, as the design of the output unit evolved, it became apparent that the interrupt controller would fit in as an extension of the output unit. The basics of the interrupt controller are similar to how the Sentinel system determines the red, green, and blue components for watchspace assessment. The numeric information acquired from the fuzzy logic computational unit is quantized into one of four possible interrupt levels. Each of these interrupt levels cause the Sentinel system to take some predefined actions. Table 3.1 describes each of the interrupt levels and the actions associated with them.

Table 3.1

Sentinel Interrupt Levels

Level	Priority	Form Displayed	Actions Taken
4	None	None	None
3	Standby	Activity Message	Activity Message displayed about watchspace
2	Warning	Warning: Attach Preference Message	Warning message displayed with question, Yes answer indicates attachment to watchspace, No answer indicates do nothing
1	Go	Attachment Message	High priority message displayed, Immediately attach to indicated watchspace

The messages indicated in Table 3.1 are forms message panels that require acknowledgment or choice selection from the user. Interrupt levels 1 and 3 just require an acknowledgment from the user to perform the task described in the message. Interrupt level 2 requires the user to make a choice of either going to the indicated Sentinel

watchspace or doing nothing. Attachment to the Sentinel watchspace is described in subsection 3.3.3. Doing nothing means to no action is taken by the Sentinel to move the current view of the user.

There is also the added capability of a sound message being played when interrupt levels 1 and 2 activate. The implementation subsection 4.3.6 on the Sound Class further defines these sound messages.

The last area, virtual keyboard control, is more of an input/output task then just an output task. The virtual keyboard allows for user input with just the mouse or any other screen driven input device. The virtual keyboard could come in handy with applications that employ the BOOM2M™ or a Head Mounted Display (HMD) and some type of data glove or flying mouse. Development of the virtual keyboard with this application is intended for a future use than with the current state of the SBB.

3.3.5 Computational Unit

The computational unit is the work horse of the Sentinel system. It is here that the Sentinel system processes all the necessary data collected. All the other units either tie into or receive information from the computational unit (see Figure 3.2). The flow of information is from the input unit into the computational unit and back out to the control and output units.

The computational unit uses fuzzy logic as its means of determining watchspace assessments for the Sentinel. Simulation information is fed into the fuzzy logic software for each Sentinel watchspace. This information includes the numbers and types of players currently in each watchspace as well as other information such as watchspace size and friend to foe ratio. The Sentinel system converts these inputs into fuzzy variables and processes them against a set of decision rules to come up with a watchspace assessment value for each Sentinel watchspace. After this concludes, the watchspace assessment information on each Sentinel watchspace passes to the control and output units and is then

displayed to the user. Chapter four has further information about the fuzzy logic implementation within the Sentinel system.

3.4 Class Structure

As mentioned above, the Sentinel system is not a pure object-oriented system. However, it is developed with a class hierarchy that encapsulates all the previously defined structured programming units. In this way, provided the programmer uses just the class methods provided, a programmer can develop an object-oriented "like" system. The following subsections address only the classes provided by the Sentinel system. This thesis does not address application (SBB) dependent classes.

There are two classes used by the Sentinel system. We use the first class for encapsulation of most of the structured programming units, initialization, and computational interfacing with the driving application (SBB). This class is called the FL_Sentinel Class. We use the other class for manipulation of the Sentinel player object with encapsulation of the control unit that directly controls the Sentinel players. This class is called the FLS_Player Class. Figure 3.6 shows the class structure and hierarchy for the Sentinel system. The following subsections discuss these aforementioned classes and their methods in a general way. Detailed information about the classes method calls and variables can be found in the programmers manual in Appendix II.

3.4.1 FL_Sentinel Class

The FL_Sentinel Class derives from the Attachable_Player Class that derives from the Player Class (see Figure 3.6) of the ObjectSim application framework developed by Capt. Mark Snyder ([Sny93]).

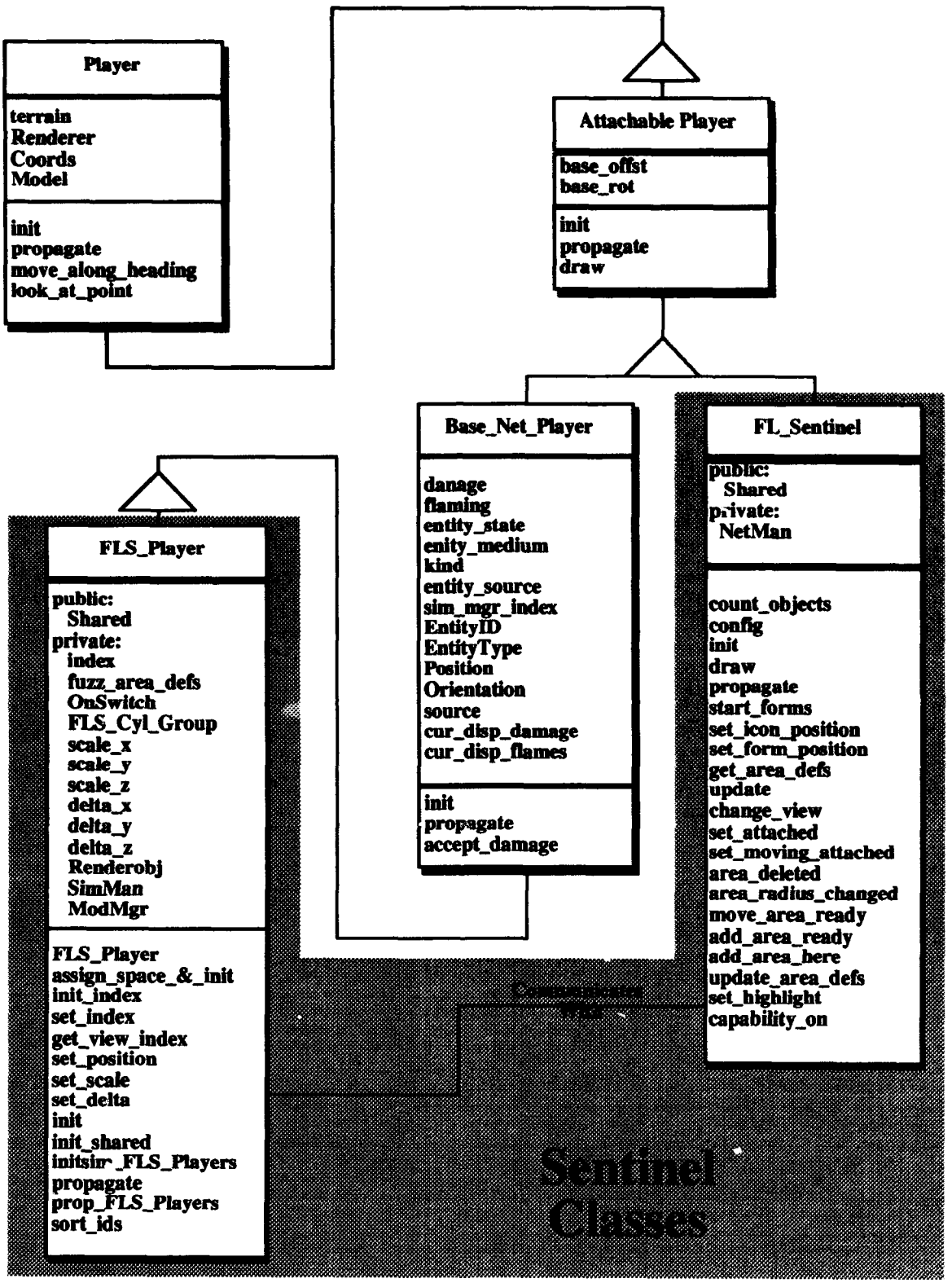


Figure 3.6: Class Structure For The Sentinel System.

The FL_Sentinel Class is the overall communication module for the Sentinel system. The class represents the main interface between the application (in this case the SBB), the FLS_Player Class, and the structured programming units. The design of the FL_Sentinel gives the driving application access to the procedures and functions needed to configure and initialize the Sentinel system. The class provides a method for the application that allows for the transfer of information about the current simulation state to the fuzzy logic computation unit of the Sentinel system. The class also provides a communication capability with the FLS_Player Class to allow the user to manipulate and modify the Sentinel watchspaces as required. The class also allows for the visual presentation of Sentinel watchspace entity capability information without going through the FLS_Player Class. Finally, the class provides a method that allows the application to output watchspace assessments computed by the fuzzy logic computation unit to the output unit for presentation to the user. The following paragraphs discuss the design of the aforementioned functional areas. Chapter four addresses the actual implementation of this class.

There are a number of method calls that allow the application access to the configuration and initialization functions found in the structured programming units. These method calls also setup the shared memory needed by the Sentinel system to pass information between control threads associated with Performer ([McL92]). The function of these method calls allows the Sentinel system to setup based upon configuration information located in configuration files.

There are three types of configuration information needed by the Sentinel system. The first type deals with the rule base needed by the fuzzy logic computation unit. The second type deals with Sentinel watchspace placement. The last type deals with player entity and designation capability information. Currently the Sentinel is setup to only take into account entity weight information for capability (relative numbers between 0.0 and 1.0

for overall capability). This could also include various other performance factors that would combine to give a more accurate capability measure. Once the Sentinel system reads the configuration information into the system and places the information into the appropriate data structures, initialization of the Sentinel system continues. The initialization amounts to setting up the proper placement and configuration of the various user control panels associated with the Sentinel system. It also allows the application to read in the current state of the simulation to initialize the fuzzy logic computation unit.

The methods that allow for the computation of the Sentinel watchspace assessments are of two types. The first type allows for the transportation of the Sentinel watchspace information over the Performer control threads. These methods allow for changes to the initial watchspace configurations that can then propagate to the needed control thread for processing and visual presentation. The second type accounts for the actual processing needed to provide information to the fuzzy logic computation unit. The method provided for this function calls the various procedures located in the input unit. The procedures are called in such an order as to produce the desired results outlined for the input unit in subsection 3.3.2.

Due to the framework of Performer, which does not allow for reading of information across threads except by way of shared memory, information transfers from the control unit to the FLS_Player Class through the shared memory allocated in the FL_Sentinel Class. This shared memory is mostly flags and index numbers into the global data structures to reflect user changes of the Sentinel watchspaces. Basically, the software sets flags and indexes in the FL_Sentinel Class and uses them in the FLS_Player Class. On the basis of these flags and indexes, the Sentinel takes certain actions on the appropriate control threads within the scope of the FLS_Player Class. The next subsection on the FLS_Player Class explains this further.

Other methods allow for the presentation of visual information within the control threads of the FL_Sentinel Class. One method allows for the presentation of visual information not directly coupled with the presentation of the actual Sentinel player as described in the FLS_Player Class or the output unit. There are currently two areas of visual presentation that the FL_Sentinel Class directly controls. The first is the visual representation of a two dimensional Sentinel watchspace cursor that gives the user an idea of the size and placement of that Sentinel watchspace. The control unit functions that take advantage of this visual cue are the following: moving an existing Sentinel watchspace and adding a new Sentinel watchspace. The second area is the visualization of entity capabilities within a Sentinel watchspace. It is here that the Sentinel system actually determines and presents to the user a visual presentation of entity capabilities within a Sentinel watchspace. A flag set from the control unit into shared memory controls this activation.

Finally, there is a method that allows for the transfer of computed watchspace assessments from the computational unit through the FL_Sentinel Class and out to the output unit for visual display to the user. A time variable that resides inside the method, controls this periodically. Currently hardwired within the code, the time variable could easily change to a configuration input or a user modifiable parameter that the user could alter during the running of the program.

3.4.2 FLS_Player Class

The FLS_Player Class derives from the Base_Net_Player Class that derives from the Attachable_Player Class that is derived from the Player Class (see Figure 3.6) of the ObjectSim application framework developed by Capt. Mark Snyder ([Sny93]).

The FLS_Player Class handles all functionally for the Sentinel player within the simulation (a player that has no physical presence in the world, only a transparent "volume" with some meaning). Functionally means how the Sentinel system renders the transparent

volume in the scene. Any modifications to the rendering parameters of the Sentinel player must be taken care of by method calls from this class or setting shared memory parameters that the FLS_Player Class can read from inside its own methods. This class manages the visual representation of the Sentinel player. The design functionally takes into account creation and initialization; placement, scale, and movement (if any) in the scene; geometric representation in the scene; and rendering order within the Performer tree.

Creation and initialization establish the initial placement of the Sentinel player within the Performer rendering tree and within the simulation. It is here that the Sentinel system assigns space to each Sentinel player in the Performer tree and gives that Sentinel player a model that represents the Sentinel player in the simulation. The Sentinel system allocates shared memory for the location of each Sentinel watchspace and sets a default scale for the size of the geometric representation based on initial configuration parameters.

The Sentinel system controls placement, scale, and movement of the Sentinel player with method calls that modify the particular Sentinel player attributes. Each time the Sentinel player goes through the propagate loop, the Sentinel system evaluates these attributes, and makes the appropriate changes as to the size and placement of the Sentinel player within the scene.

The Sentinel system can also control the geometric representation for each Sentinel player in the scene. By just changing the model index number used by the Model Manager (see subsection 4.3.5) associated with the particular Sentinel player, the Sentinel player's geometric representation changes to whatever model the new index number references. Therefore under the current design, the Sentinel player can have as many geometric representations as desired. However, note that the contained-in function determines what falls within a Sentinel player watchspace. If we chose a different geometric shape, we must call (or develop) the appropriate contained-in function to insure that the Sentinel system only counts entities that fall within the bounds of the new geometric representation.

However, if we only require modifications to just the basic attributes of the geometric representation (i.e., color, missing polygons, etc.), then we can accomplish this easily and without modifying the contained-in function.

The last important aspect of the FLS_Player Class is the ability of the class to place the Sentinel players in such a way in the Performer rendering tree that the Sentinel system renders them last and in reverse sorted order based on distance from the current view point. The reason for this is to ensure that each transparent Sentinel player volume can see all other objects, including other Sentinel players. Performer handles transparencies in a manner so that whatever renders last can see everything rendered before it. Therefore, at every frame the Sentinel system removes the Sentinel players from the Performer tree, sorts them in reverse order according to distance from the current view point, and then reinserts them back into the Performer tree. Also, as mentioned earlier, the Sentinel system must render this "branch" of the Performer tree last after all other transparent objects in the tree.

The FLS_Player Class along with the FL_Sentinel Class make up the class structure used by the Sentinel system. It is through these classes that the Sentinel system uses the structured programming units and maintains the encapsulation of their existence.

3.5 Conclusions

The overall design of the Sentinel system takes into account both the object-oriented class structure methodology and structured programming techniques. It mixes these two approaches in such a way as to come up with a design that is both easy to attach to a driving application and user friendly. The mix is such that the programmer, using the Sentinel class structures for a particular application, never need know that the design is not pure object-oriented. All the programmer need ever know is where to place the method calls within their code to activate and use the Sentinel system.

The next chapter deals with the actual implementation of the Sentinel system applied to the SBB. The chapter addresses the data structures used and the system integration with

the SBB as well as other supporting frameworks and tools. The chapter also discusses the use of stand alone utilities designed especially for the Sentinel system, as well as, Sentinel system operation.

IV. System Implementation

4.1 Introduction

This chapter looks at the implementation of the Sentinel system within a large scale synthetic environment. This large scale synthetic environment goes by the name of the Synthetic BattleBridge (SBB). The SBB uses the Sentinel as an extension to enhance situational awareness. This chapter looks at some of the implementation issues involved in adding the Sentinel system to the SBB.

Five sections make up the rest of this chapter dealing with implementation. Section 4.2 examines the implementation decisions associated with the various data structures of the Sentinel system. Section 4.3 addresses the issue of integration of the Sentinel system with other major frameworks, applications, and toolkits that comprise the Sentinel and SBB system. Section 4.4 discusses some Sentinel system utilities created for the configuration of the Sentinel system environment. Section 4.5 describes the operation of the Sentinel system within the framework of the SBB. Finally, section 4.6 presents some conclusions about the overall implementation process.

4.2 Data Structures and Implementation Decisions

The following subsections look at the various data structures used in the Sentinel system. Some of these data structures are global and available to the entire Sentinel system. The following subsections point out where the Sentinel system initializes and uses these global data structures. The subsections also talk about some of the implementation decisions associated with the data structures themselves.

4.2.1 Configuration Unit

The computation unit handles most of the initialization of the global data structures needed by the Sentinel system. The main global data structure used by nearly every

structured programming unit and class of the Sentinel system, called current_config, carries current information about the state of the Sentinel system. This state information includes the current number of active Sentinel watchspaces, the positions of those watchspaces, the maximum values associated with each of the fuzzy logic categories needed by the computation unit, and a current list of players located within the Sentinel watchspaces.

Another global data structure, called obj_weight_array, holds the current entity weight information for each of the predefined DIS entity types and designations required by the Sentinel system. A header file, created by a Sentinel system utility, enumerates the predefined DIS entity types according to a configuration file supplied by the user. The configuration file, called Object_Types.dat, contains a list of all the DIS entity types required and the category they belong in as designated by the user for the current simulation run. This enumerated list supplies various array structures with easy access to their array elements for processing.

Figure 4.1 shows how the user can modify these entity weights using the "Fuzzy Category Configuration" control panel. The user activates this control panel by pressing the "Config Sentinel" button on the screen. Subsection 4.2.3 explains the implementation and use of this icon button. With this control panel, the user can choose from three main categories of entities or objects. Each main category divides into six sub-categories. As shown in Figure 4.1, the lit buttons indicate where the user made changes to the entity or object weights. These buttons stay lit until the user saves this information using the "Save" or "Save & Hide" buttons. The "Cancel & Hide" button returns without saving any changes. The user also has the ability to load in a predefined object weight file by pressing the "Load" button. The user can also make all the objects have the same level of importance by pressing the "Reset 1.0 Weights" button. Finally, the user can see, but not change, the current DIS protocol representation that the Sentinel system currently uses (flat or round earth).

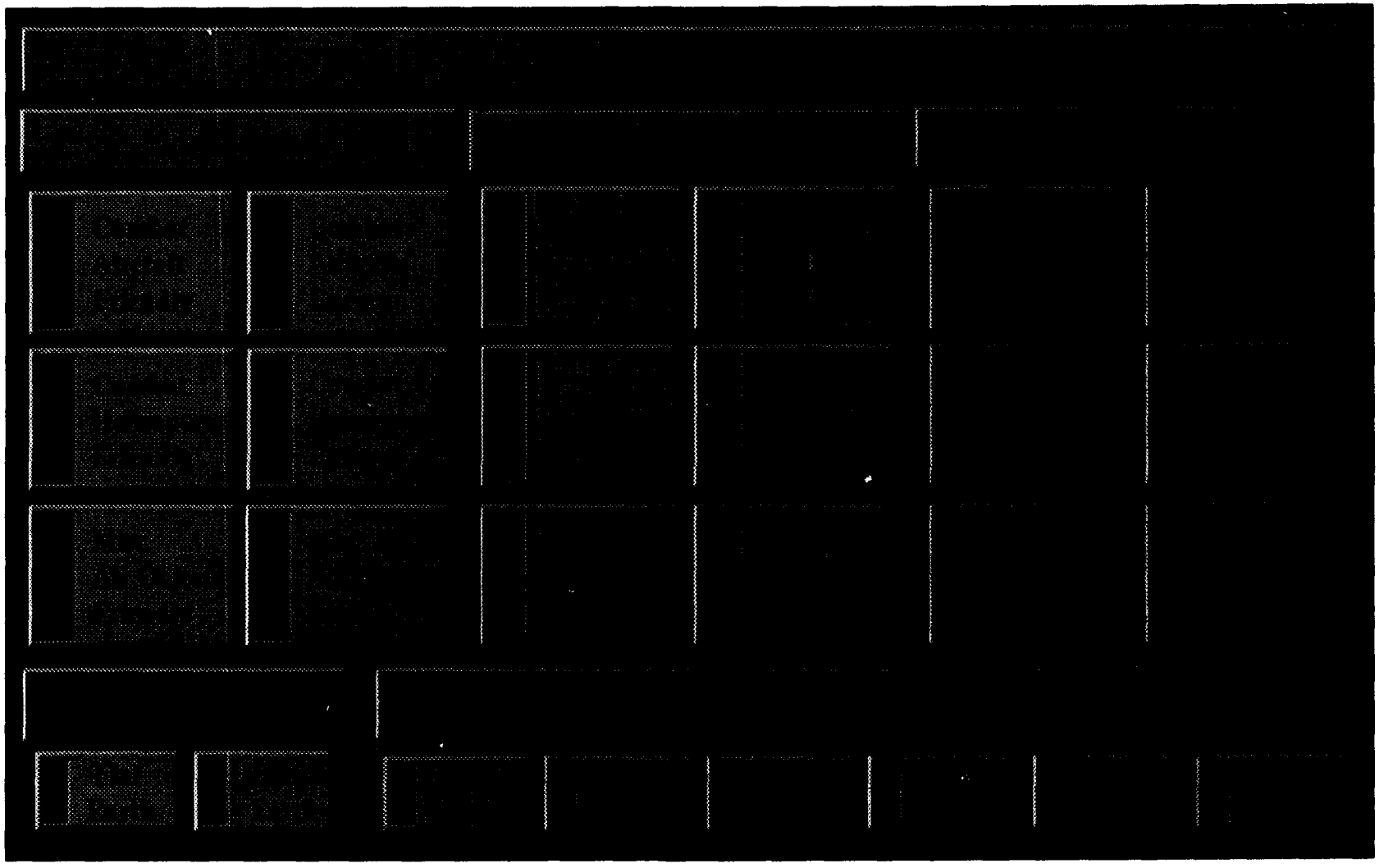


Figure 4.1: Fuzzy Category Configuration Control Panel

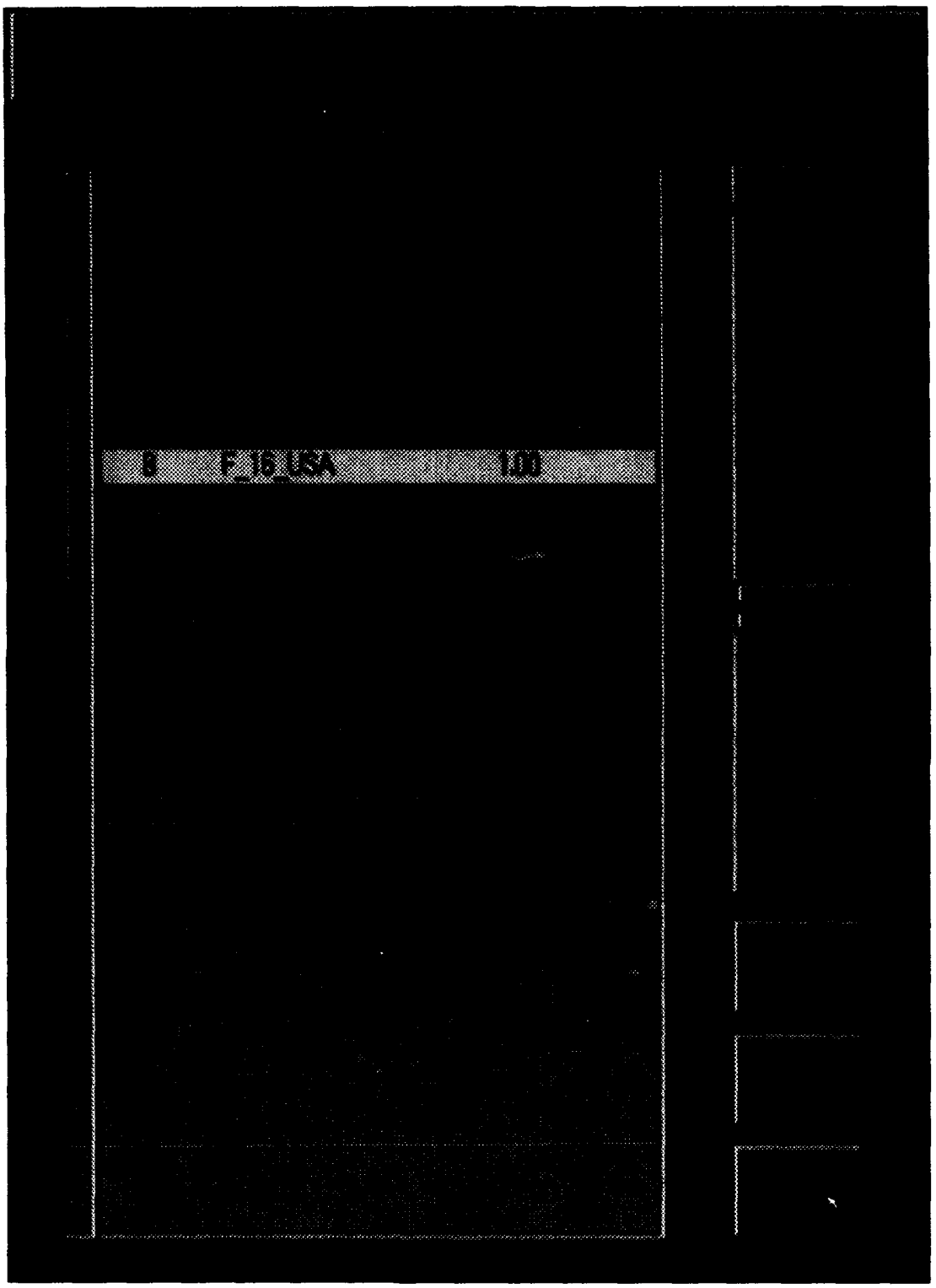


Figure 4.2: Category Weight Browser and Editor

When the user selects one of the sub-category buttons, the user sees a display as shown in Figure 4.2. The user can change weight settings for a particular entity or object here. Figure 4.2 shows that the user is about to changed the weight of the F_16_USA designation (entity) from 1.00 to 0.42 The user selected this entity from a sub category called "Combat Aircraft Friendly" that also is a sub category of "Air Objects".

Another implementation issue associated with the configuration unit deals with converting positional information into a flat earth or round earth representation. A flag in the configuration file called default_areas.dat indicates whether the data converts to flat earth (DIS_FE) or round earth (DIS_RE) representation. Conversion to flat earth requires a local origin at position 0,0,0 with a lat-long position. The Sentinel system then interpolates the new x-y-z position from this local origin to the lat-long center of the Sentinel watchspace. The system can also use an absolute x-y-z position for the Sentinel watchspace from a configuration file called xyz_FE_default_areas.dat. This allows for the direct input of x-y-z position information as opposed to lat-long position information. In this case, the system ignores interpolation and uses the x-y-z position as absolute position relative to the local origin. Conversion to round earth requires that the system converts lat-long information to round earth x-y-z position data via the World Geodetic System 1984 (wgs84) database. Currently the system can use both flat and round earth methods. However, implementation with the SBB requires that Sentinel watchspaces conform to the x-y-z flat earth representation. This happens because the SBB converts all incoming round earth position information into flat earth position information. Therefore, since the SBB contains the Sentinel system, the Sentinel system must communicate only x-y-z flat earth position information.

4.2.2 Input Unit

The main function of the input unit is to collect and process information about the types and numbers of player entities within each Sentinel watchspace. To do this the input

unit uses a number of global data structures. The input unit uses two global data structures initialized by the configuration unit, one global data structure intended for use in the computation unit, and one internal data structure used by the unit itself.

The input unit takes information about each Sentinel's position and size from the Sentinel's state global data structure initialized by the configuration unit (current_config). As the system sorts each entity within each Sentinel watchspace bound, the system loads a count array for each different type of entity found within the Sentinel watchspace. Once the system totals this information for each Sentinel watchspace, the system processes the count arrays based on the required information needed by the fuzzy logic computation unit. The system then stores this processed information in a global data structure (Incoming_FLS_Data). The input unit then passes this global data structure to the computation unit.

For the current implementation of the Sentinel, the fuzzy logic computation unit requires a percentage of how many entities of a certain entity group are there within in each Sentinel watchspace. Currently the fuzzy logic computation unit requires 18 different entity group percentages, as well as, the watchspace size in square miles for each Sentinel watchspace. Subsection 4.2.5 talks more about the implementation of the fuzzy logic computation unit.

4.2.3 Control Unit

The main purpose of the control unit is to manage the various user interface control panels and to translate user input from those panels to a visual representation within the scene.

The implementation of the Sentinel system's user interface hinges on the number of control panels displayed to the user based on the current state of the Sentinel system. Currently the Sentinel system can manage five levels of user interface control panels. Each of these levels allows the user a different level of control over the Sentinel system. We

identify these five control levels as follows: Icon level, Low Detail level, High Detail level, Attached Control level, and Functional Control level. The following paragraphs address each of these levels.

The Icon level gives the user the most basic control over the Sentinel system. The Sentinel system presents the user with two icon buttons, currently displayed in the lower left hand corner of the screen. However, the programmer can choose to place these buttons anywhere on the screen by using a method call from the FL_Sentinel Class. Each button represents one of two main user interface control panel paths. The button with the file and hand icon on it with the "Config Sentinel" text, opens up the main user interface control panel for configuring entity weight values as described in subsection 4.2.1 (see Figure 4.1 above). The other button with the eye and magnifying glass icon on it with the "Sentinel" text, opens up the Low Detail level associated with actual control and viewing of Sentinel watchspaces. Figure 4.3 shows these icon buttons in the lower left hand corner of the screen. Figure 4.3 also shows some Sentinel watchspaces in the current screen view. Note that Figure 4.3 shows approximately 1/2 the actual screen width and height.

The Low Detail level gives the user some added visual information about each Sentinel watchspace along with some limited functionally and viewing capabilities. The user can see the watchspace assessment bar associated with each Sentinel watchspace and its indicated value. Subsection 4.2.4 describes how the Sentinel system determines these colors and associated bar lengths for each Sentinel watchspace. The Low Detail level also gives the user the ability to attach or detach from any given Sentinel watchspace. The paragraph on High Detail level addresses how attachment or detachment occurs. Note this Low Detail level uses watchspace slot numbers for the names of the Sentinel watchspaces. By doing this, we can keep the control panel small for casual viewing. However, the user must make the connection between the slot number and the actual Sentinel watchspace name. The buttons on the bottom of the control panel give the user access to either a higher

or lower level of control from this point. The button labeled "Hide" reverts to the Icon level of control while the button labeled "More" gives the user access to High Detail level control panel. Figure 4.4 shows the Low Detail level control panel in the lower left hand corner of the screen. Figure 4.4 also shows some Sentinel watchspaces in the current screen view. Note Figure 4.4 shows approximately 1/2 the actual screen width and height.

The High Detail level gives the user more visual information and functionally than does the Low Detail level. The user sees a bigger control panel along with two visual pieces of information not available with the Low Detail level. First, the user now sees the first nine characters of the Sentinel watchspace's name. Therefore, if the user chooses the Sentinel watchspace's name carefully, the user can make a quick connection between the name presented in the slot and where the Sentinel watchspace resides. The second added piece of visual information shows the user at a glance what the current interrupt level is for each watchspace. The watchspace assessment status bar now shows the actual interrupt level text associated with each Sentinel watchspace. Figure 4.5 shows the High Detail level control Panel. Once again, note that Figure 4.5 shows approximately 1/2 the actual screen width and height.

A new functional button at the bottom of the control panel allows the user to add new Sentinel watchspaces to the simulation provided the availability of a Sentinel watchspace slot. When the user presses the "Add" button, the Sentinel system presents the user with a Functional level control panel. This control panel instructs the user on how to add a new Sentinel watchspace to the simulation. Other functionals associated with the High Detail level allows the user once again to access other levels of control. The "Hide" button performs the same function as described in the Low Detail level. However, the "Prev" button allows the user to go back to the Low Detail level control. The other functionally allowed at this level is the ability to attach or detach from any Sentinel watchspace.

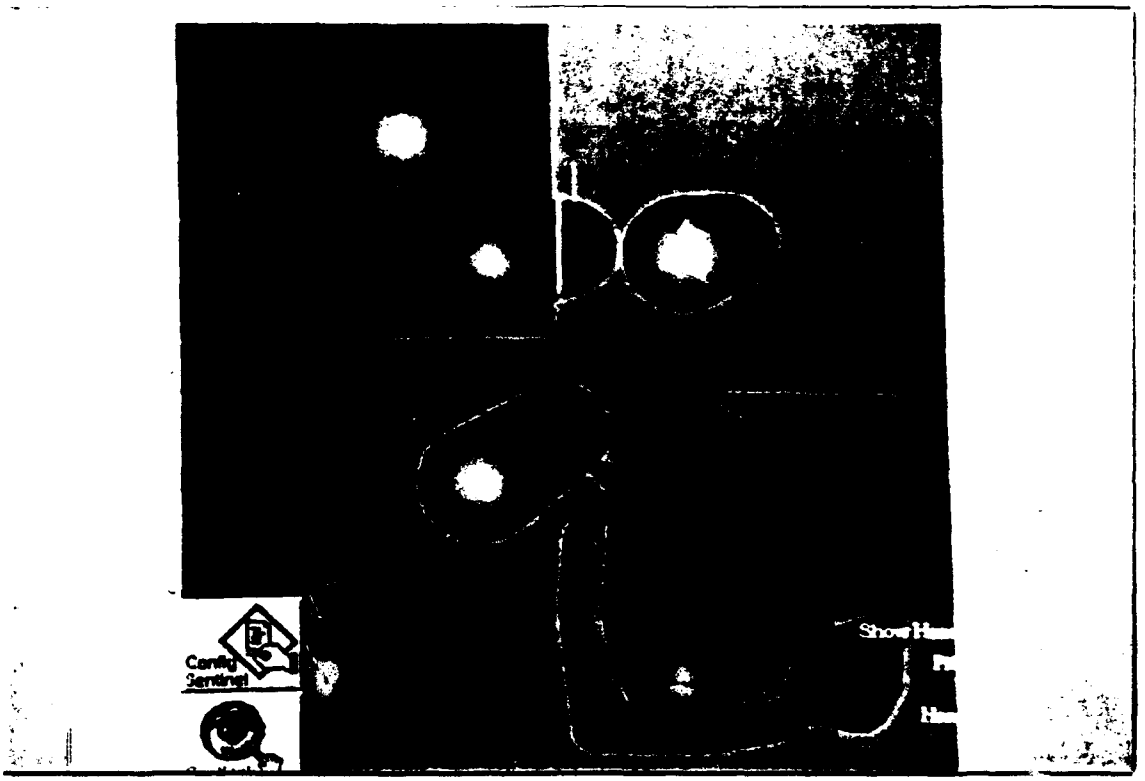


Figure 4.3: Icon Level Control Panel.

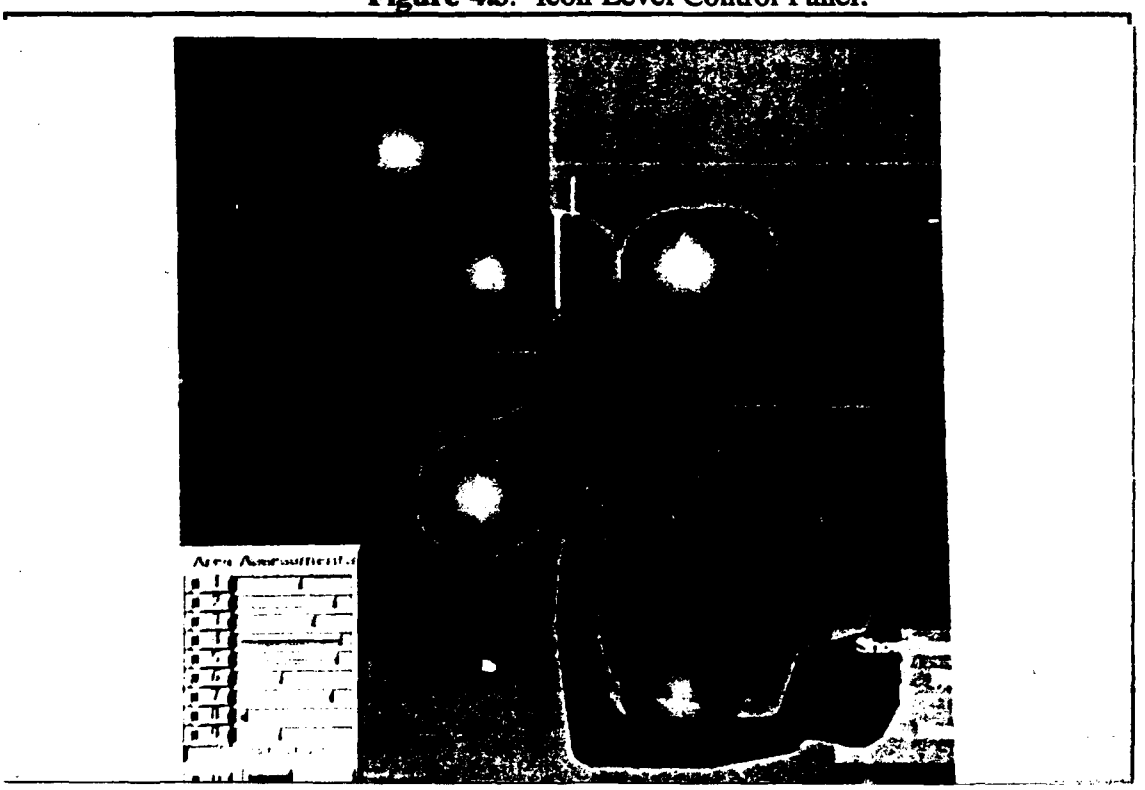


Figure 4.4: Low Detail Level Control Panel.

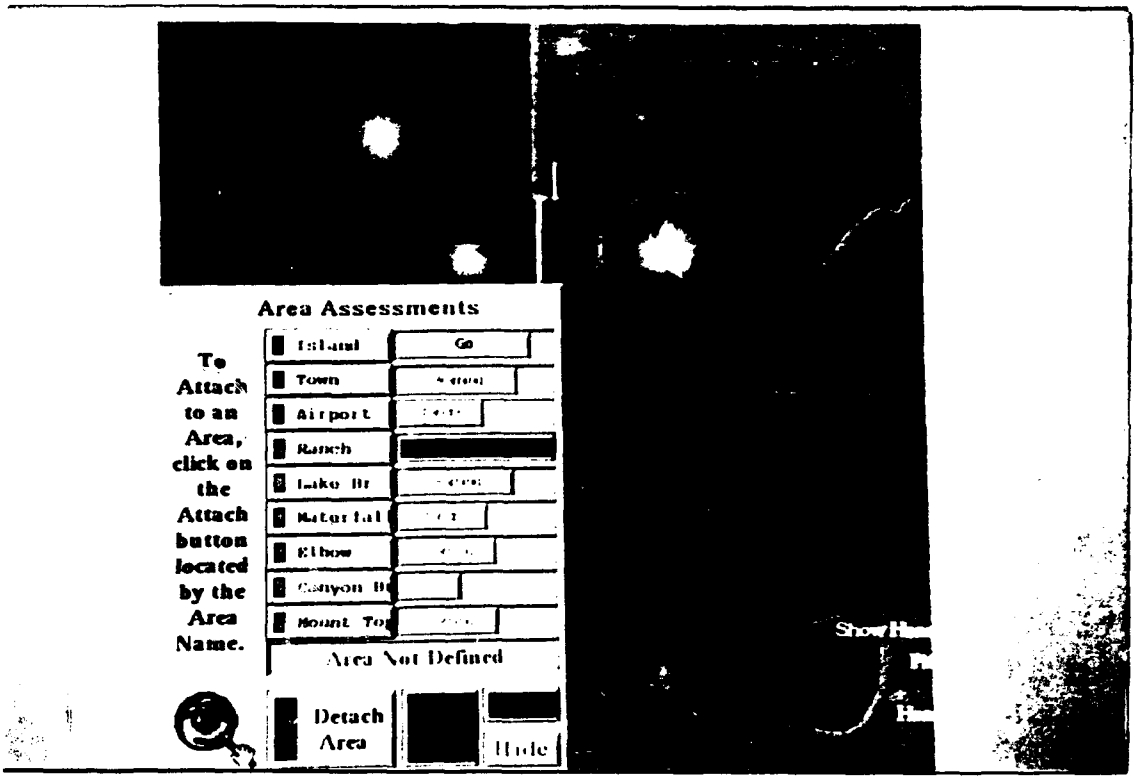


Figure 4.5: High Detail Level Control Panel.

To attach to any Sentinel watchspace, all the user has to do is to push the appropriately named Sentinel watchspace button. When this happens, the detach button light goes out and the pushed Sentinel watchspace button goes on. Attachment to a Sentinel watchspace causes the view to switch from the current view to the view for that Sentinel watchspace as described in subsection 3.3.3. There are currently three scales of attach viewing for Sentinel watchspaces (1.25, 5.0, and 10.0). The scale used depends on the function being performed. A standard attachment uses a 1.25 scale (STANDARD_ALT_SCALE) of the actual height needed to fit all the Sentinel watchspace base just onto the screen. Once the user pushes an attachment button, the Sentinel system goes to the Attached Control level as described next.

The Attached Control level (see Figure 4.6) gives the user the ability to manipulate the Sentinel watchspaces directly. Once attached to a Sentinel watchspace, the Sentinel System presents the user with six buttons that can manipulate the Sentinel watchspace or can view and correlate Sentinel watchspace information. The button's names are as follows: Move, Modify Radius, Capability Contour, Show History, Reset View, and Delete. The buttons themselves set flags and/or Sentinel watchspace index numbers in the FL_Sentinel Class shared memory structure. On the basis of these flags and/or index numbers, the FL_Sentinel and FLS_Player Classes react accordingly. Subsections 4.2.6 and 4.2.7 discusses this shared memory structure and how it applies to the Sentinel watchspaces based on user input. With the exception of the six functional buttons mentioned above, everything else is almost the same as the High Detail level control panel. The only other difference is that now the "Detach" button indicates not only detaching from the current Sentinel watchspace, but also allows the user to go back to the High Detail level control panel.

Figure 4.6 shows the Attached Control level control Panel. Figure 4.6 indicates that the user attached to a Sentinel watchspace named "Airport." We can tell this by the lit attached button next to the name "Airport." This control panel is similar to the High Detail level control panel, with the exception that now the control panel has a row of six functional buttons to the left of the attachment buttons. Also, Figure 4.6 shows the user located in the center of the attached Sentinel watchspace. The unshaded circle area is the base of the Sentinel watchspace and the shaded part indicates looking out through the transparent inside of the Sentinel watchspace volume (a cylinder in this case). Also, note that now Figure 4.6 shows nearly all the height of the screen and about 5/6 of the width of the screen. This is why the control panel appears smaller then in Figure 4.5 above.

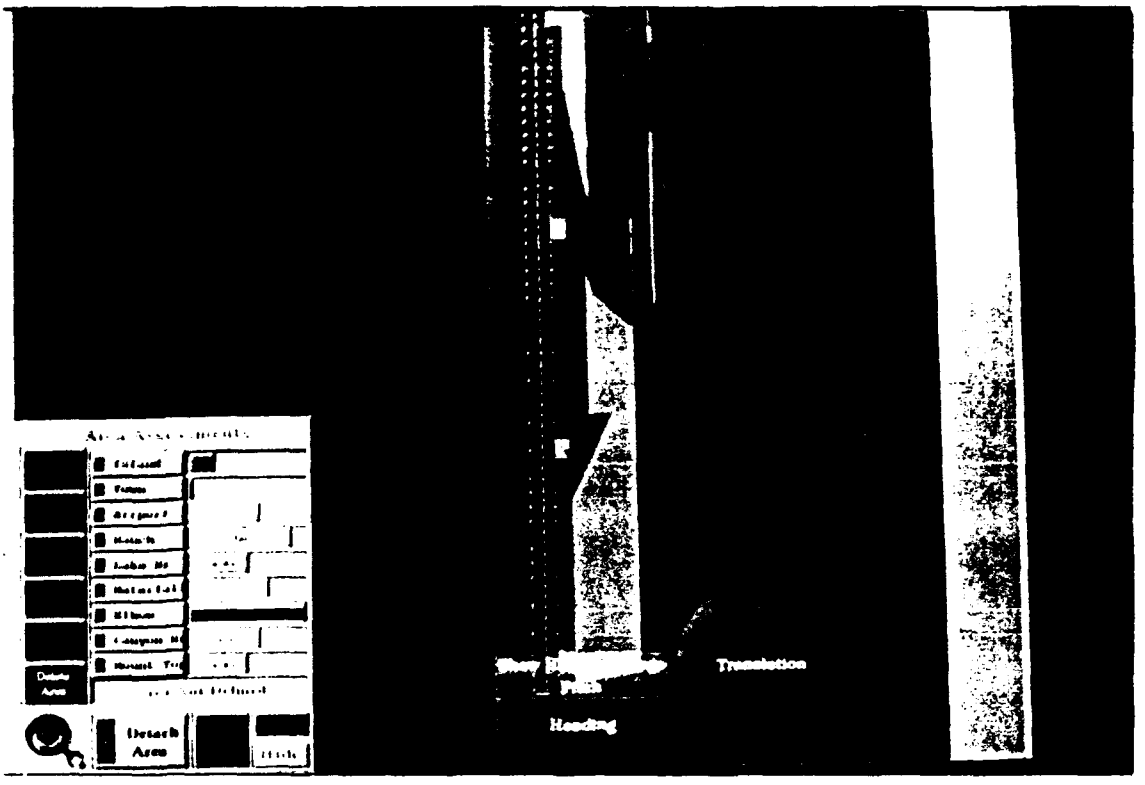


Figure 4.6: Attached Control Level Control Panel.

The last user interface control panel level, Functional Control, gives the user instructions and control over the more complex functions allowed by other user interface control panels. Currently there are five Sentinel functions handled by different Functional Control levels. The functional areas are as follows: Fuzzy Category Configuration (Weight Values), Show Watchspace History, Modify Watchspace Radius, Add Watchspace, and Move Watchspace. Subsection 4.2.1 talks about the Fuzzy Category Configuration (Weight Values) control panel (see Figure 4.1 in subsection 4.2.1).

The Show Watchspace History control panel (see Figure 4.7) presents the user with a strip chart of past watchspace assessments for that particular Sentinel watchspace. The use of a circular queue (array type) stores the watchspace assessments for each Sentinel watchspace. Each item in the queue holds a watchspace assessment value, a corresponding red, green, blue triple, and a color map index. Subsection 4.2.4 on the Output Unit

discusses how the Sentinel system derives the red, green, blue triple and the need for a color map index. Currently the Sentinel system can store and show 50 time slices in one chart. As new information arrives, old information moves to the left and the new piece of information gets inserted at the far right of the strip chart. Figure 4.7 shows an example of a full watchspace assessment history over time. Note that the "Return" button in the upper right hand corner of the control panel allows the user to remove the strip chart from the screen.

The Modify Watchspace Radius function has one control panel associated with it (see Figure 4.8). This control panel gives the user current information on the radius for the particular Sentinel watchspace of interest. The user can then "apply" changes to the radius and see the effect of the change in the simulation. The user then has the option to "ok" the change or "reset" the radius back to what the radius came in as. When the user invokes this function, the Sentinel system resets the view position and sets the viewing mode to plan mode. Also, since it would be hard to see the change of radius from within the Sentinel watchspace of interest (remember, attachment puts the user into the watchspace and high enough to see the full base of the watchspace on the screen), the Sentinel system temporary places the user at the mid altitude scale (MID_ALT_SCALE) of 5.0 times the normal attachment height above the scene. In this way, the user can see the effect of the radius change on the surrounding area. Once the user completes the radius change (if any), the view returns to the standard attachment view for the Sentinel watchspace with the radius changes (if any) incorporated. Figure 4.8 shows the Modify Radius control panel. Notice the highlighted Sentinel watchspace to the left of the control panel. This visually shows the user what Sentinel watchspace the radius change effects.

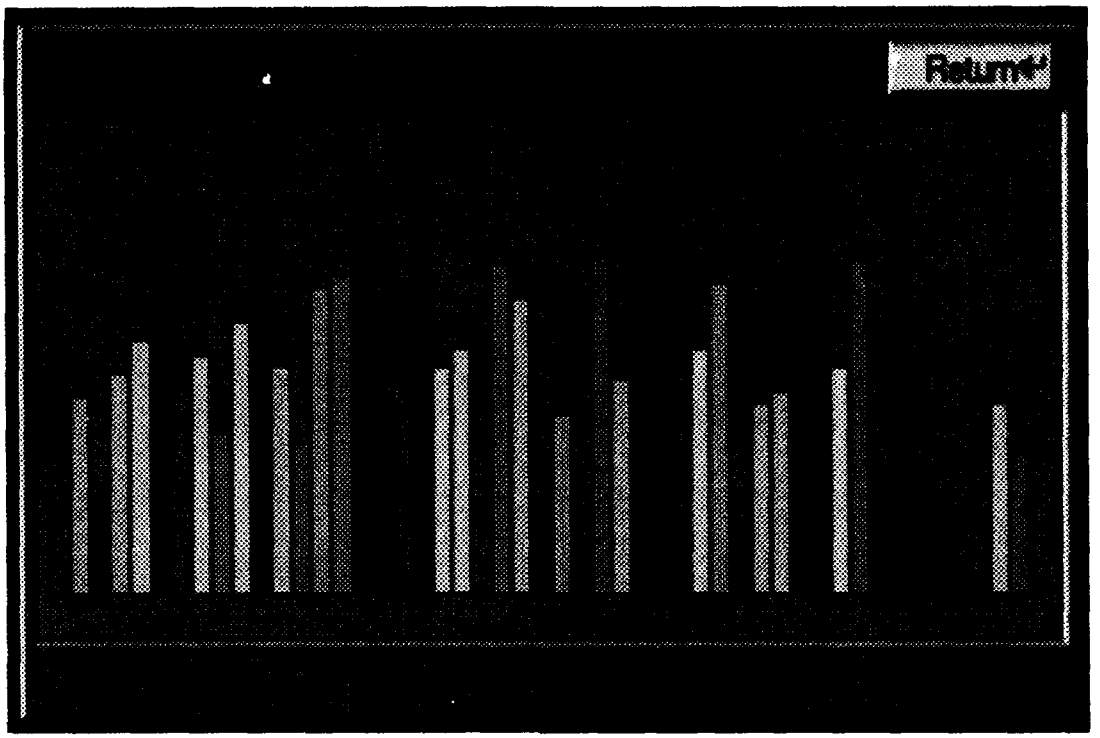


Figure 4.7: Sentinel Watchspace Assessment History Strip Chart

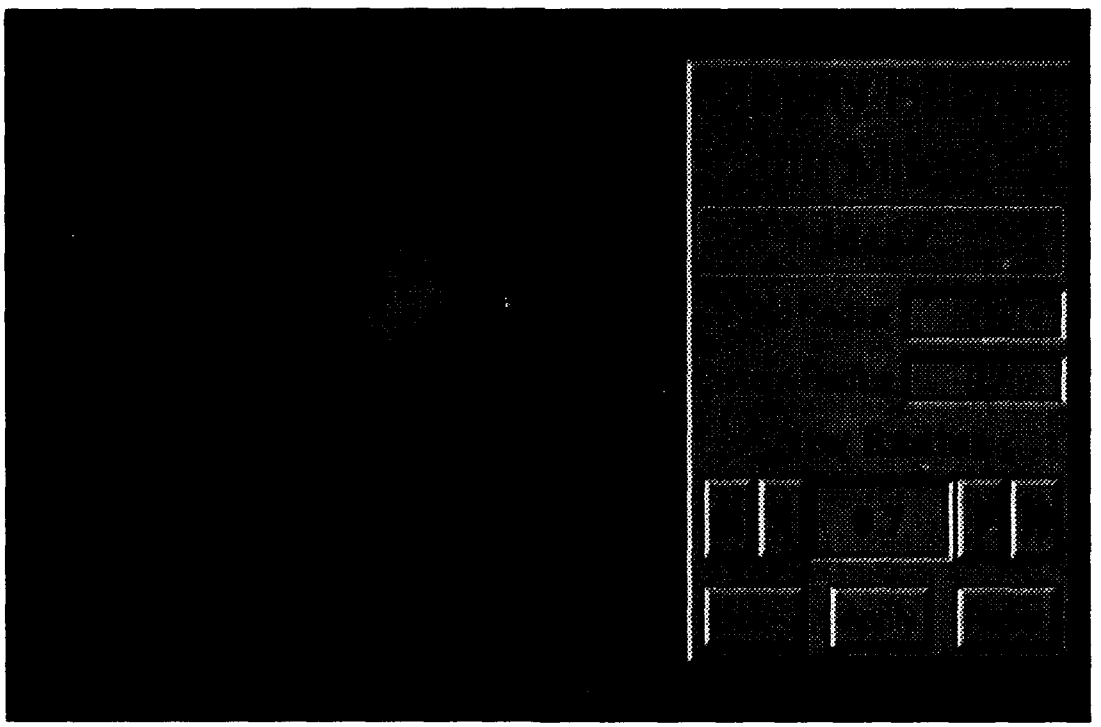


Figure 4.8: Modify Radius Control Panel

The Move Watchspace and Add Watchspace functions (see Figures 4.9 - 4.13) behave in much the same way. The difference is that the Move Watchspace function works with an already defined Sentinel watchspace, while the Add Watchspace function creates a temporary Sentinel watchspace to move and place in the simulation. Both functions cause the Sentinel system to reset the view position (for Add Watchspace, the view position is the center of the terrain) and set the viewing mode to plan mode. Just like the Modify Watchspace Radius function, the Sentinel system temporary changes the view height so that the user can better see the placement of the Sentinel watchspace. The Sentinel system uses the high altitude scale (HIGH_ALT_SCALE) of 10.0 times the normal attachment height above the scene. For both functions, the Sentinel system presents the user with a pre and during control panel. The pre control panel gives the user instructions on how to place or move a watchspace in the scene. When the user moves to where the Sentinel watchspace position is to be and presses "ready" two things then happen.

First, the control panel changes to the during phase that has new instructions and different control buttons. Second, the Sentinel system changes the cursor to a transparent disc that is the same size of the Sentinel watchspace base being moved or a default size if adding a new Sentinel watchspace. The disc also has a cross hair on it along with the x and y position relative to the scene. Subsection 4.2.6 discusses how the FL_Sentinel Class implements this disc cursor. Once the user determines the new position for the watchspace, the user can "position again", "undo", "reset", or "accept" the Sentinel watchspace placement. When the user completes the move, the view transfers to this new location with the standard attachment view. One additional thing happens when the user adds a new Sentinel watchspace to the simulation. After the user accepts the placement of the new Sentinel watchspace, the user must give this new Sentinel watchspace a name. The user can do this by using the keyboard or using the virtual keyboard displayed on the screen. The virtual keyboard allows the user to enter characters into an input area with the

mouse, that when ready, transfers to the input area for the Sentinel watchspace's name. The virtual keyboard can type both lower and upper case letters as well as numbers. The Clear button clears the input window and the Enter button transfers the given input. There is also a backspace button (blue solid triangle pointing left) that removes one character at a time. Once the user gives the new Sentinel watchspace a name, the Sentinel system reflects this new watchspace and name into the appropriate data structures and control panels.

The following figures show the Move Watchspace and Add Watchspace control panels. Figure 4.9 and Figure 4.10 show the pre- and during- control panels for the Move Watchspace function respectively. Figure 4.11 and Figure 4.12 show the pre- and during-control panels for the Add Watchspace function respectively. Notice how the transparent disc in Figure 4.12 is smaller then the transparent disc in Figure 4.10. However, the radii of both discs are the same size. This is because when we are adding a Sentinel watchspace to the simulation, we are twice as high as we would be if we were just moving an existing Sentinel watchspace. Therefore, the projection of the circular area from the view point to the ground of the higher view point appears smaller on the ground. The final figure in this subsection, Figure 4.13, shows the virtual keyboard and the Change Watchspace Name control panel associated with adding a new Sentinel watchspace.

4.2.4 Output Unit

As mentioned in subsection 3.3.4 on the design of the output unit, the implementation of the output unit breaks up into four functional pieces or areas. The grouping of these functional areas are as follows: initialization and forms display control, display of Sentinel watchspace information, interrupt control, and virtual keyboard control. Clearly some of these functional areas are both output and control areas. However, the following paragraphs on the individual functional areas address the design and implementation decisions on grouping them in the output unit.

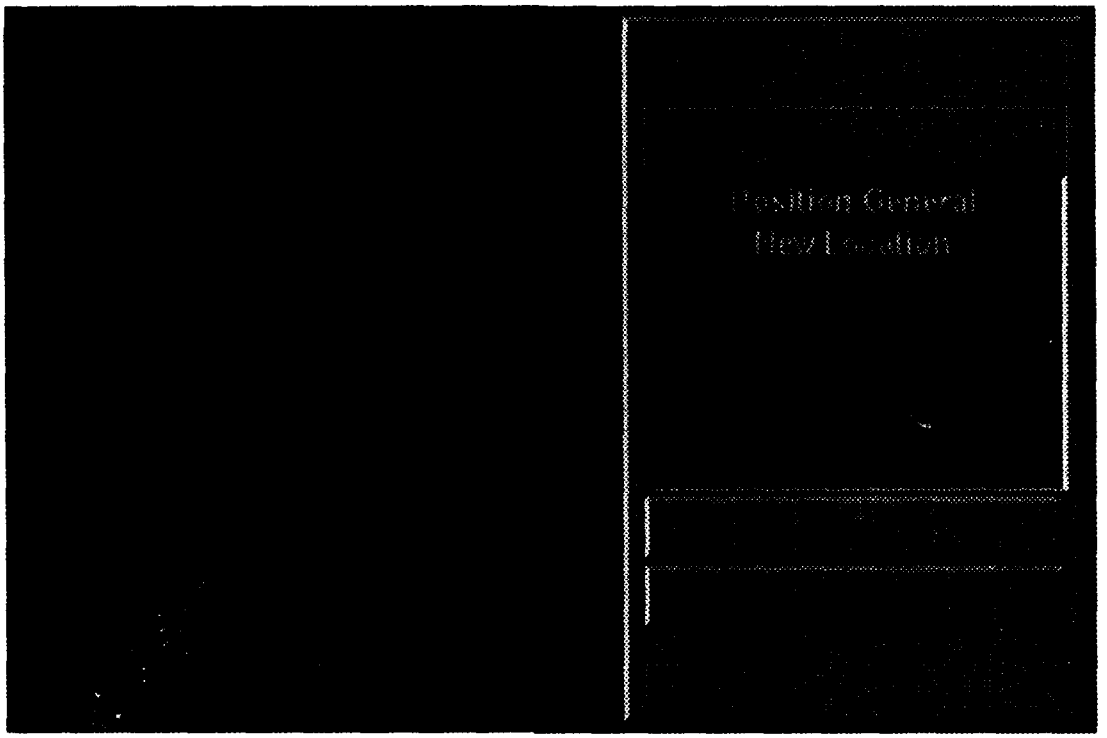


Figure 4.9: Move Sentinel Watchspace Pre Control Panel

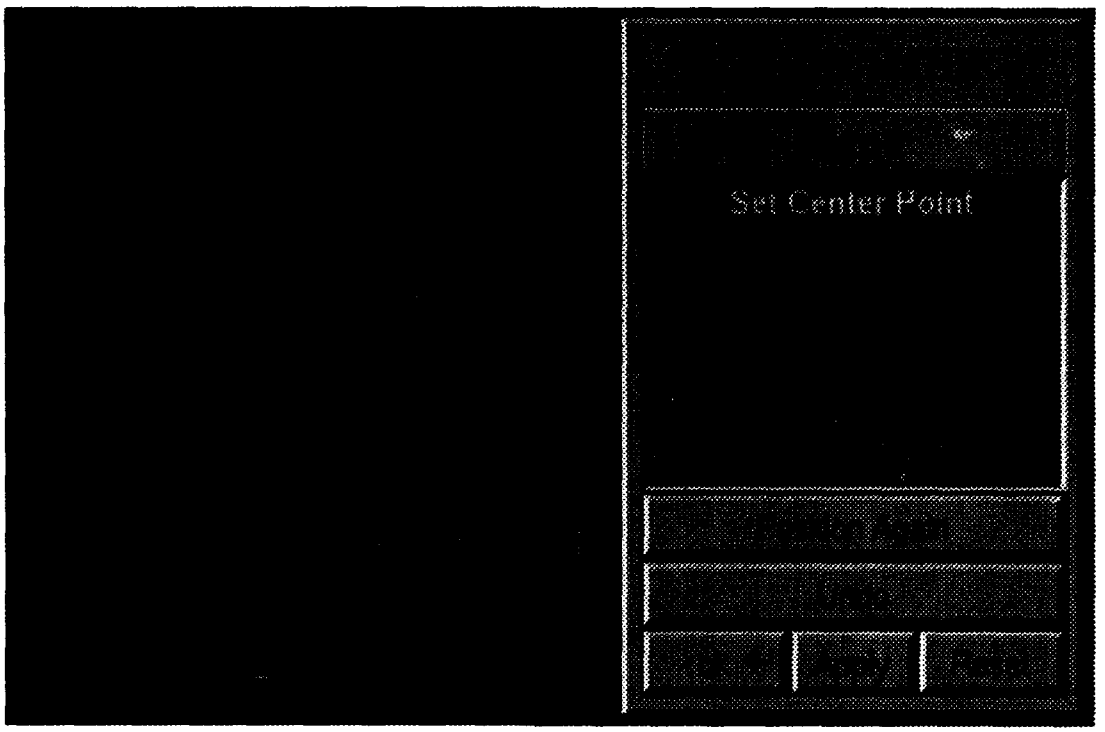


Figure 4.10: Move Sentinel Watchspace During Control Panel

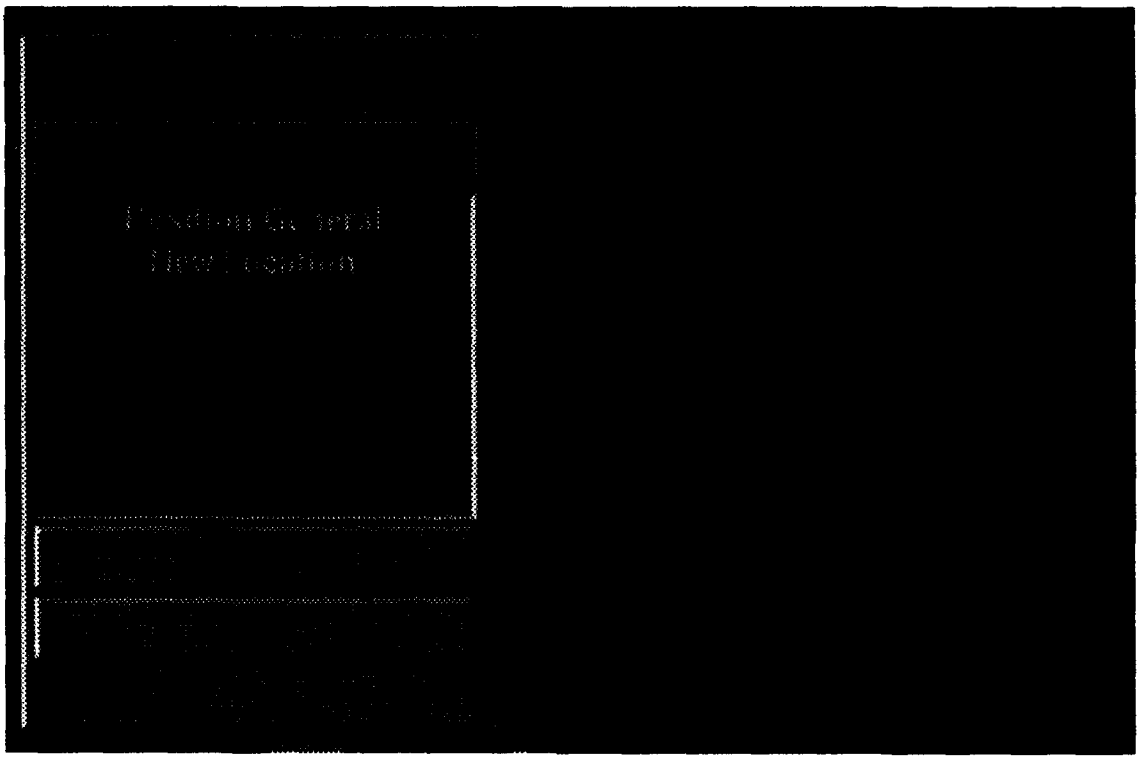


Figure 4.11: Add Sentinel Watchspace Pre Control Panel

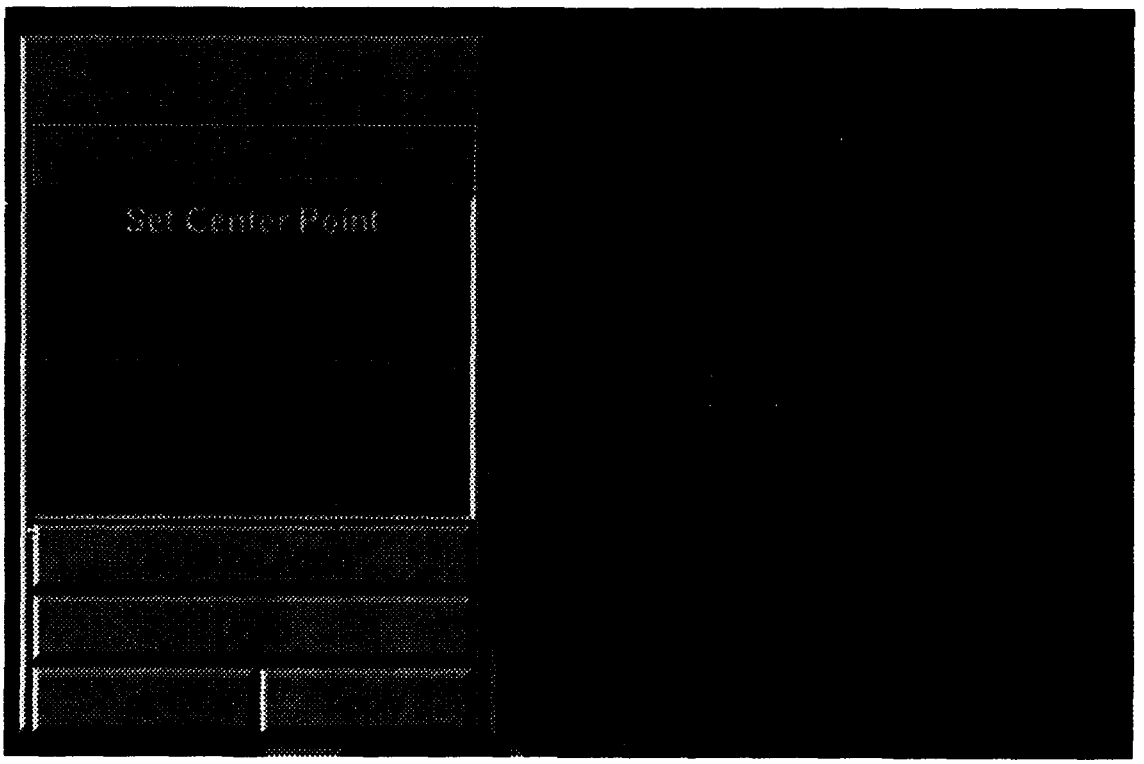


Figure 4.12: Add Sentinel Watchspace During Control Panel

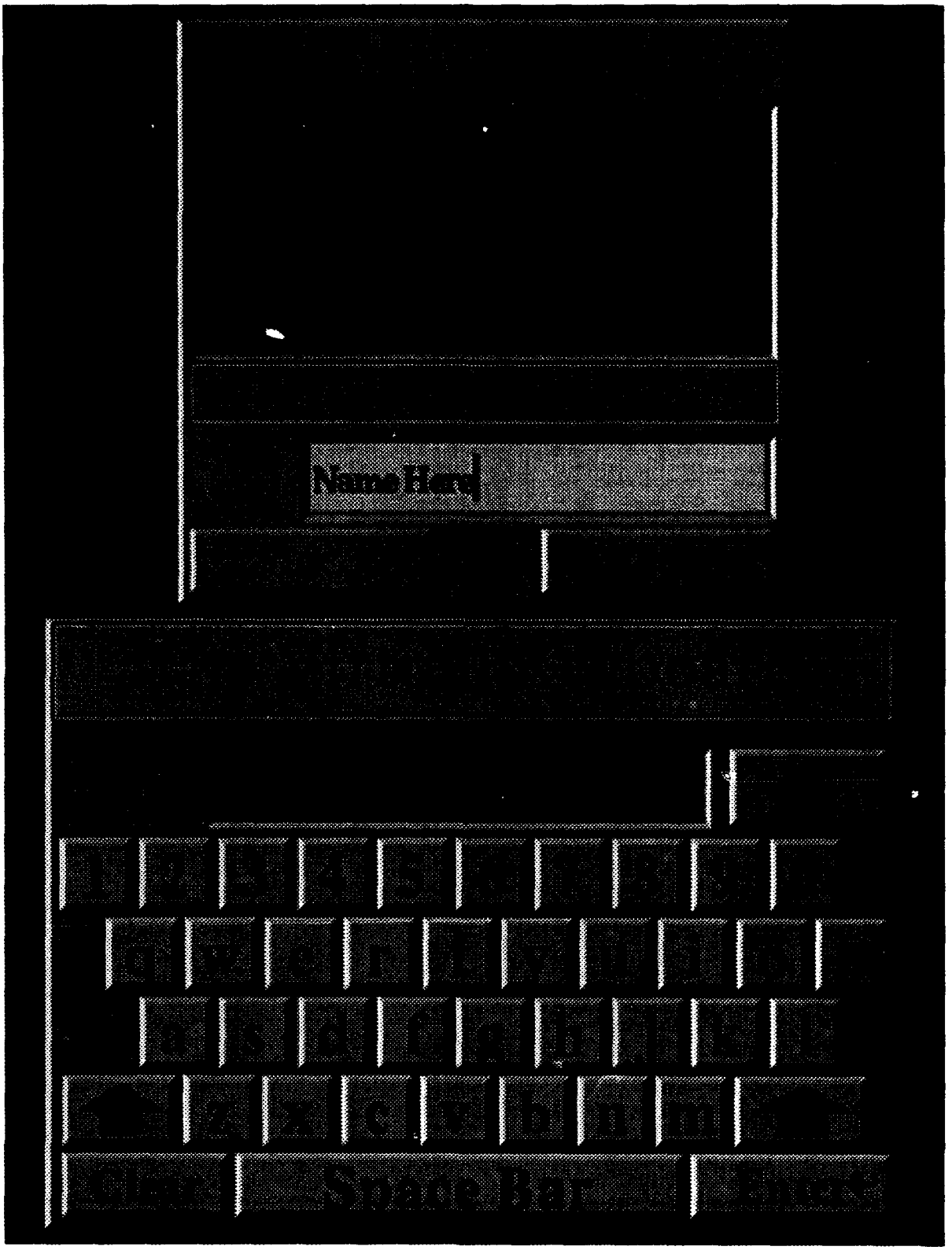


Figure 4.13: Virtual Keyboard and Change Watchspace Name Control Panel

The first function area of implementation of the output unit deals with initialization and forms display control. Initialization means setting up all the correct initialization information needed for the display of all the various user interface control panels associated with the Sentinel system. The Sentinel system does this by using the global data structures initialized by the configuration unit. In particular, the Sentinel system uses the number of Sentinel watchspaces and Sentinel watchspace names to setup the watchspace assessment control panels. The second part of initialization deals with setting up and displaying the user interface control panels. The construction and display of the control panels use the Forms 2.1 Library (see subsection 4.3.1). Therefore, there are a number of initialization steps needed to set up and display the control panels. Each control panel represents a "form" that contains a number of "form objects" that make up the form. We must initialize each of these form objects before the application can use them. The Sentinel system does this by one procedure call, Output_Form_Constructor, that builds all the control panels specified. Once specified, other procedures set the form objects to certain values for start up. These form objects also get updated by the Sentinel system during run time.

Interrupt control is the second functional area of the output unit. Interrupt control resides with the output unit because interrupt control causes the Sentinel system to possibly change the view that the user sees based on the interrupt level. The current implementation uses four levels of interrupt (see Table 3.1). Each interrupt level has an associated name that appears centered on the watchspace assessment status bar for each Sentinel watchspace. As shown in Table 3.1, the interrupt levels may have associated user control with them. Table 4.1 shows the current watchspace assessment ranges and interrupt level names used with the four interrupt levels. Note that currently the Sentinel system uses pre-programmed interrupt level ranges. However, we could modify this so that the user could define the interrupt level ranges in a configuration file. The Sentinel system would read in

this file at start up and set up the parameters appropriately. Also, it would not be hard to have the system modify these ranges during run time.

Table 4.1

Sentinel Interrupt Ranges

Name	Low Watchspace Assessment Value	High Watchspace Assessment Value	Display Actions
None	0.00	< 0.45	None
Standby	0.45	< 0.60	Centered on watchspace status bar
Warning	0.60	< 0.80	Centered on watchspace status bar
Go	0.80	1.00	Centered on watchspace status bar

The third function area of the output unit deals with presenting the watchspace assessments made by the computation unit to the user in a visual form. By using the Forms 2.1 Library we can use sliders to represent values relative to one another ([Ove92]). In other words, if we set the bounds of the sliders to handle values between 0 and 1 then a longer slider would indicate a value closer to 1 then a shorter slider. If we add to this a color coding that has a one-to-one mapping between a red, green, and blue triple and a slider value, then we can present to the user a visual cue. The user can then quickly compare this visual cue against other similar cues and come up with a comparative rating between all the cues. Therefore, we present the user with two methods of comparison that relate to one another. The first one is a comparison between sliders of different lengths with a longer length relating to a higher watchspace assessment value (risk).

The second one is the mapping of color codes to higher watchspace assessment value (risk). The Sentinel system does this by mapping the watchspace assessment value obtained from the computational unit to a red, green, and blue triple that displays a certain

color. The current implementation of the color coding goes from low to high value (risk) as follows: blue, light blue, blue green, green blue, yellow green, green red, orange, light red, medium red, and red. An association here could be that little or no risk relates to a blue color, medium risk relates to a green yellow color, and high risk relates to red color. Figure 4.14 shows a bar that goes from a value of 0.0 at the left to a value of 1.0 at the right. At any place along the bar, the color at that position represents the value at that position. The three graphs above the bar indicated how much of each color component (0 - 255) makes up the color triple at any value from 0.0 to 1.0.

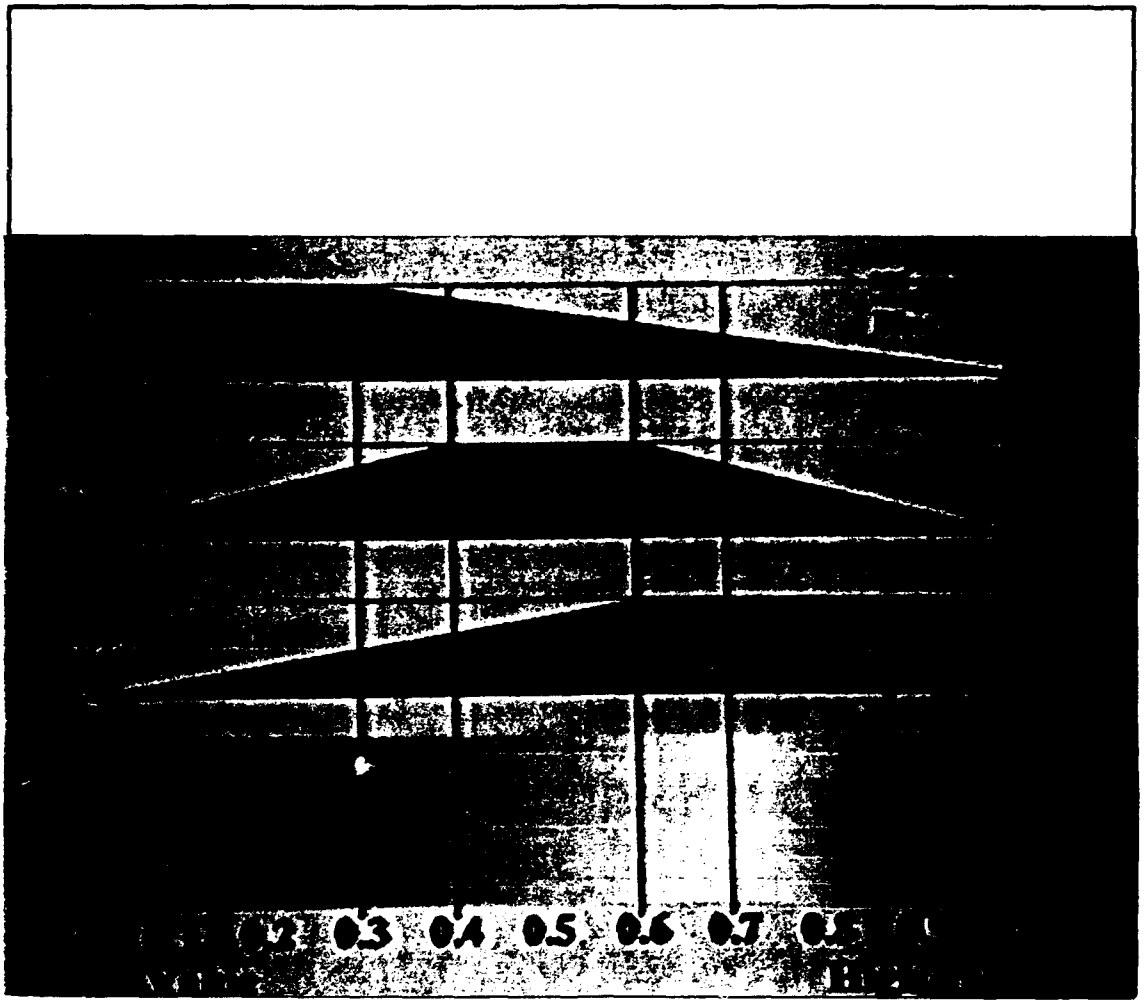


Figure 4.14: Sentinel Watchspace Assessment Status Bar Color Components

The last functional area of the output unit controls the display of the virtual keyboard. It is here that the virtual keyboard displays all the characters in the correct case depending on whether or not the user pushed the shift button located on the virtual keyboard. Also, it is here that user input through the virtual keyboard buttons (keys), get transferred to the output window of the virtual keyboard. Finally, the control of transferring the input from this window to the input window of the appropriate control panel occurs within the output unit.

4.2.5 Computation Unit

The computation unit houses the heart of the Sentinel system. It is from here that all the other units and classes interact directly or indirectly. The black box ties to the computation unit are simple. There exist two global data structures for which information passes into the computation unit and passes out of the computation unit. The configuration and input units determine what the computation unit needs from the current state of the simulation. This information passes into the computation unit. The computation unit then passes out watchspace assessment information that the output unit then uses to visually represent each Sentinel watchspace to the user. For design, this black box approach simplifies the mechanism of the Sentinel system. The implementation of the support mechanisms just needs knowledge of the particulars of the into and out of global data structures. The support systems process the needed input data to the computation unit and then visually process the output data from the computational unit. Therefore, the design and implementation of the inside of the black box computational unit determines how realistically the Sentinel system mimics human intelligence gathering in the field. The following paragraphs discuss these issues and the choice of using fuzzy logic to mimic human thought processes.

We have completed our initial investigation into the usefulness of the fuzzy logic control paradigm (see [Kin77], [Kos92], [Lee90], [Mam74], [Zad73]) to assist a virtual

environment user in assessing interesting activity and automatically moving the commander to an interesting portion of the battlespace at the appropriate time . We chose to use fuzzy logic because it can recognize a pattern of activity and mimic human judgments concerning the significance of the patterns. Because fuzzy logic allows us to assess the relative importance of an input in relation to other inputs, the system can adaptively react to changes in an environment. This characteristic effectively duplicates a human's response to environmental changes. We adapted the fuzzy logic controls paradigm to the problem of assisting, informing, and automatically positioning a user in a virtual environment by developing a fuzzy logic assistant, which is at the heart of the Sentinel, to monitor and assess activity within the battlespace.

We have implemented the first version of the Sentinel. This implementation uses a simplified model of the battlespace and was designed to determine the usefulness of fuzzy logic and of our approach to improving situational awareness. The initial implementation is based upon the process model outlined in Figure 4.15. To enable us to assess interest for specific areas of the battlespace, we allow the user to interactively place Sentinels throughout the battlespace, one for each desired watchspace. Each Sentinel operates identically. The following discussion describes the operation of a single Sentinel.

The Sentinel provides the user with a visual signal indicating the appropriate level of interest for a watchspace (output unit). The level of interest is a numerical representation of the information that would be sent to the commander by a tactical operations center based upon observations by scouts. Each Sentinel checks on the status of its watchspace during each assessment cycle. At the beginning of each cycle, the Sentinel determines the numbers and types of vehicles, troops, and other important information within its watchspace (input unit).

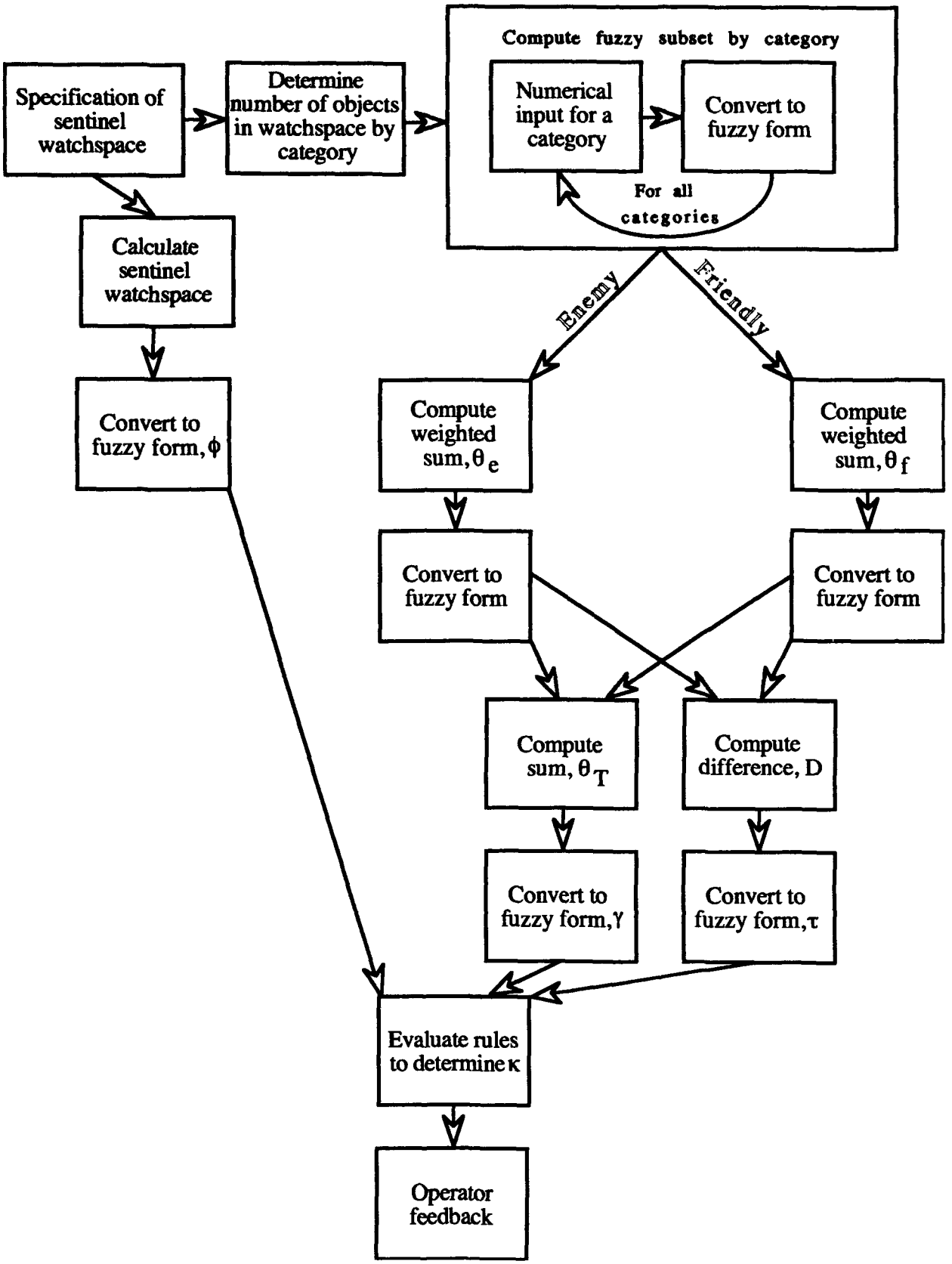


Figure 4.15: Process Model for Computing Interest Level for a Sentinel's Watchspace.

These inputs convert into fuzzy variables, after which we evaluate decision rules to give an overall assessment of the appropriate level of interest for the watchspace. This operation of the Sentinel is described later. The result of each assessment is communicated to the commander using a visual display (output unit). The display allows the commander to remotely monitor the overall activity within selected watchspaces without moving to an observation point for the watchspace or trying to assess the activity for the entire battlespace.

The procedure used by the Sentinel determines the appropriate level of interest, κ, for the Sentinel's portion of the battlespace. The model we used employed general categories for types of threats within each Sentinel's watchspace of the battlespace. These categories for both friendly and enemy formations are combat aircraft threat, combat helicopter threat, infantry threat, armor threat, guided munition threat, and artillery threat. Let i represent each category. Then T(i) is the term set of i, with each value being a fuzzy number defined on the universe of discourse. Let O be an object in the battlespace. The Sentinel begins by observing the state of the fuzzy system and finds the number of objects belonging to each threat category X_i within its volume, V_s.

$$X_i = (\Sigma\, O_i \mid (O_i \text{ within } V_s)) \;\forall\, i \qquad (1)$$

The total number of objects for each category is then assigned a membership function value for each of the term sets (Big, Medium, Small, and Low). The membership functions, $\mu_i(x_i)$, within each T(i) are based on subjective evaluations and each fuzzy set is convex and normal (as in Figure 4.16).

The next step in processing is to determine the linguistic variable of X_i with the highest membership function value, ω_{x_i}. Let A, B, C, D be the four fuzzy sets associated with X_i. Equation (2), the union operation, is applied across all four fuzzy sets to determine the value of ω_{x_i}.

$$\omega_{x_i} = \mu_{(AU(BU(CUD)))}(x_i) \text{ where } \mu_{AUB}(x_i) = \max\{\mu_A(x_i), \mu_B(x_i)\} \quad (2)$$

The output from this step is then used to compute a numerical estimate for military presence for enemy and friendly forces, θ_e and θ_f, within the Sentinel's volume. Military presence is computed by taking a weighted sum of all the ω_{x_i} for each force, and falls in [0, 1].

$$\theta = \sum_{i=1}^{n}(\omega_{x_i} \Sigma p_i) \text{ where } p_i \text{ is the weighting factor for } \omega_{x_i} \quad (3)$$

The total military presence, $\theta_t = \theta_e + \theta_f$, is then assigned a membership function value for each linguistic variable within $T(\theta_t)$ and application of the approach in equation (2) results in the γ value for presence. The membership functions for θ_t are based on subjective evaluations and each fuzzy set is convex and normal.

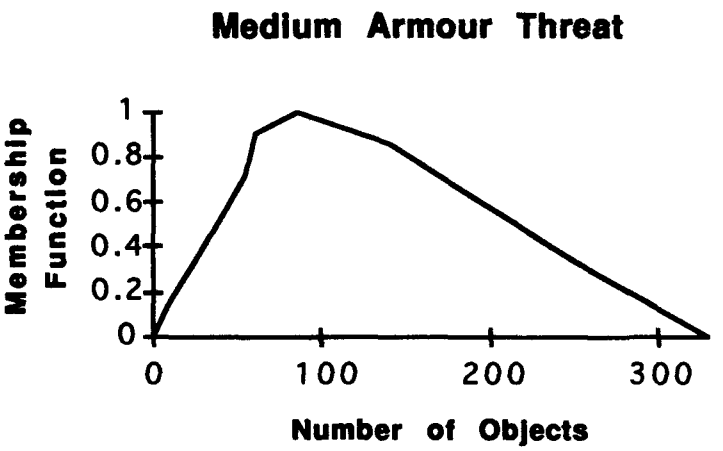


Figure 4.16: Fuzzy Set Defining a Medium Threat by Armor.

Two other variables are used to evaluate the level of interest in the Sentinel's watchspace; the size of each watchspace, ϕ, and whether the enemy forces outnumber the friendly forces, τ. The first step in determining ϕ is computing the projected area, A, of the

watchspace being monitored by a Sentinel. The projected area, A, has a term set, T(A), and the linguistic variable of A with the highest membership function value is found by applying the approach described in equation (2).

We determine the outnumbers variable, τ, by looking at the difference, D, between enemy and friendly forces within the Sentinel's watchspace. That difference is the input to the membership function evaluation of T(D), (Yes, No, Equal), with the value of τ determined by applying the approach we presented in equation (2).

The values for τ, ϕ, and γ are used as input to the rule set to determine the appropriate control action. The processing up to this point serves to summarize the information within a watchspace into larger, conceptually related, information aggregates. We exploited this aggregation process when we designed the rule set so as to minimize the computational time required to evaluate the control actions. The rule set design produced 48 rules that are capable of responding to all input conditions. A few example rules are presented in Table 4.2.

TABLE 4.2

Sample Fuzzy Logic Rules

Rule 1	If γ is BIG and ϕ is SMALL and τ is YES then level of interrupt is GO and color is RED.
Rule 2	If γ is MEDIUM and ϕ is SMALL and τ is EQUAL then level of interrupt is STANDBY and color is BLUE GREEN.
Rule 3	If γ is LOW and ϕ is LARGE and τ is NO then level of interrupt is NONE and color is BLUE.

The control actions that result from the rule set evaluation process provide the commander with notification concerning the appropriate level of interest required for a Sentinel's watchspace. Feedback is provided using two general cueing mechanisms, a

sliding scale (see subsection 4.2.4) and a level of interrupt (see subsection 4.2.4). The sliding scale gives visual feedback by changing color and length according to the value of κ. In addition, there are four interrupt levels that only engage under specific circumstances. Subsection 4.2.4 addresses both of these cueing mechanisms and how the current implementation of the Sentinel system uses them in relation to the SBB.

4.2.6 FL_Sentinel Class

As mentioned earlier in the design chapter, the FL_Sentinel Class' main role is to act as a mediator between the driving application and the structured programming units. The FL_Sentinel Class accomplishes this task by setting up method calls and shared memory. The method calls act as the main go between for the structured programming units. The shared memory allows for the transfer of global data structures between Performer threads. This transfer is mainly along a path from the control unit, through the FL_Sentinel Class, and to the FLS_Player Class. The FL_Sentinel Class mainly passes data around the Sentinel system. However, the FL_Sentinel Class does have two drawing capabilities associated with its draw thread.

The first of these drawing capabilities was pointed out earlier when we discussed the implementation of the move watchspace and add watchspace commands. When either one of the aforementioned commands takes place, the cursor gets replaced by a transparent disk with red crosshairs and green terrain coordinates. Figure 4.10 and Figure 4.12 show examples of this transparent disk. The graphics library (GL) procedure blendfunction along with various other primitive GL calls made the transparent disk possible. The following paragraphs cover more on the blendfunction and its usefulness within the Sentinel system.

The other drawing capability that is possible within the FL_Sentinel Class directly deals with the conceptual visualization of player information within each Sentinel watchspace, as depicted in Figures 4.18 - 4.20. As mentioned in subsection 4.2.3 on the

control unit, one of the Sentinel system's functions allows the user to view a conceptual capability contour as it exists within a Sentinel watchspace. The idea here is to give each player entity a capability rating based on characteristics like speed, acceleration, armament, friend or foe, etc.. This capability is then mapped to a particular color and a particular radius of a slightly transparent disk. These capability disks are then placed upon a neutral background in the same position as the corresponding player entity in the Sentinel watchspace. These transparent capability disks then overlap one another on the background. These transparent disks combine on the background with the use of the GL call blendfunction. What results are transparent colors that blend together to form rough contour lines of capability. How these transparent colors blend together are a function of the parameters given to the blendfunction call.

For our implementation, we use the blendfunction with the following two parameters: BF_DC and BF_ZERO. By doing this we scale each frame buffer color component by the incoming color component with the blending function:

blendfunction (BF_DC, BF_ZERO)

Rdestination = min (255, (Rsource * (Rdestination / 255)))

Gdestination = min (255, (Gsource * (Gdestination / 255)))

Bdestination = min (255, (Bsource * (Bdestination / 255)))

Adestination = min (255, (Asource * (Adestination / 255)))" ([McL91:15-7])

The following paragraphs describe the actual color coding and implementation of the capability contour for the Sentinel system within the SBB. To begin with, we use only two attributes when calculating the capability of any player entity: entity type and entity force type. We use entity type to look up the entities weight factor as provided in the configuration files. This number is between 0 and 1, and relates the relative capability of this entity player to all others. We use this weight factor to calculate the radius of the disk

for each entity type. A weight factor of one yields us the maximum capability radius allowed by the Sentinel system. We also use the weight factor to determine the total green color component of the rgb color for the disk. The green component can vary from 255 down to 55. This green component is then part of a rgb triple where the red and blue components are predetermined based on entity force type.

If the entity force type is friendly, we use **RGBcolor(55, green_component, 200)**, else we use **RGBcolor(200, green_component, 55)** for foes. By doing this we have color ranges of yellow to orange red for foes, and color ranges of light blue to dark blue for friends. The reason we do not put the maximum values for red and blue in the above procedure calls and only go down to 55 for green is so that when overlap occurs, the blendfunction has some room to overlap the colors and produce a darker shade of red for foes and a darker shade of blue for friends. In this way, we can see how conceptually the darker areas indicate where capabilities overlap and therefore are a greater threat. The same principle applies when opposing forces overlap. We immediately see darker colors associated with this overlap that would indicate a clash of forces is eminent. Also, because the way the blendfunction behaves, the red and blue components cancel each other out to leave only the green component. In this way, the user can view the capability contour and instantly see the disposition of forces in the watchspace. Levels of blue indicate friendly forces, levels of red indicate opposing forces, and levels of green indicate overlap of both forces.

Figure 4.17 consolidates the above information. Along the top of the graph the triangular area shows the color and radii associated with the capability of friendly forces. Along the side of the graph the triangular area shows the color and radii associated with the capability of unfriendly forces. Together these mix with the use of the blendfunction to create the square greenish area that represents the amount of overlap between opposing forces.

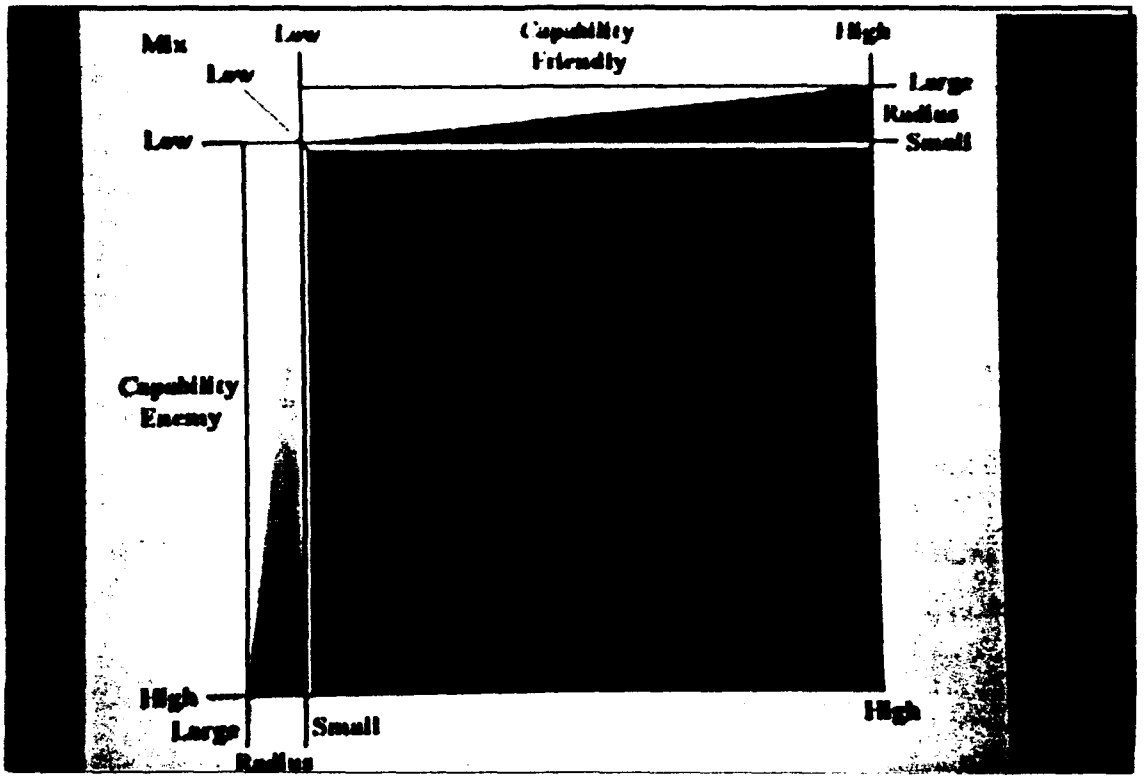


Figure 4.17: Capability Contour Color Mixing Graph

The following three figures show the capability contours of a Sentinel watchspace within the SBB. There were four different types of player entities involved with the simulation. Each of these entities had the following weights: F-15 = 0.75, F-16 = 0.45, M1 Tank = 0.25, and T80 Tank = 0.35. As shown in the figures, we can see darker areas of red or blue where overlap occurs within force types.

We can also see dark shades of green where opposing forces overlap. Also notice that the disks that are much smaller and of lighter color represent the tanks that are of a lesser weight value. We can also notice the difference between the F-15s and the F-16s by watching how fast the disks move across the area and the disks relative size to one another.

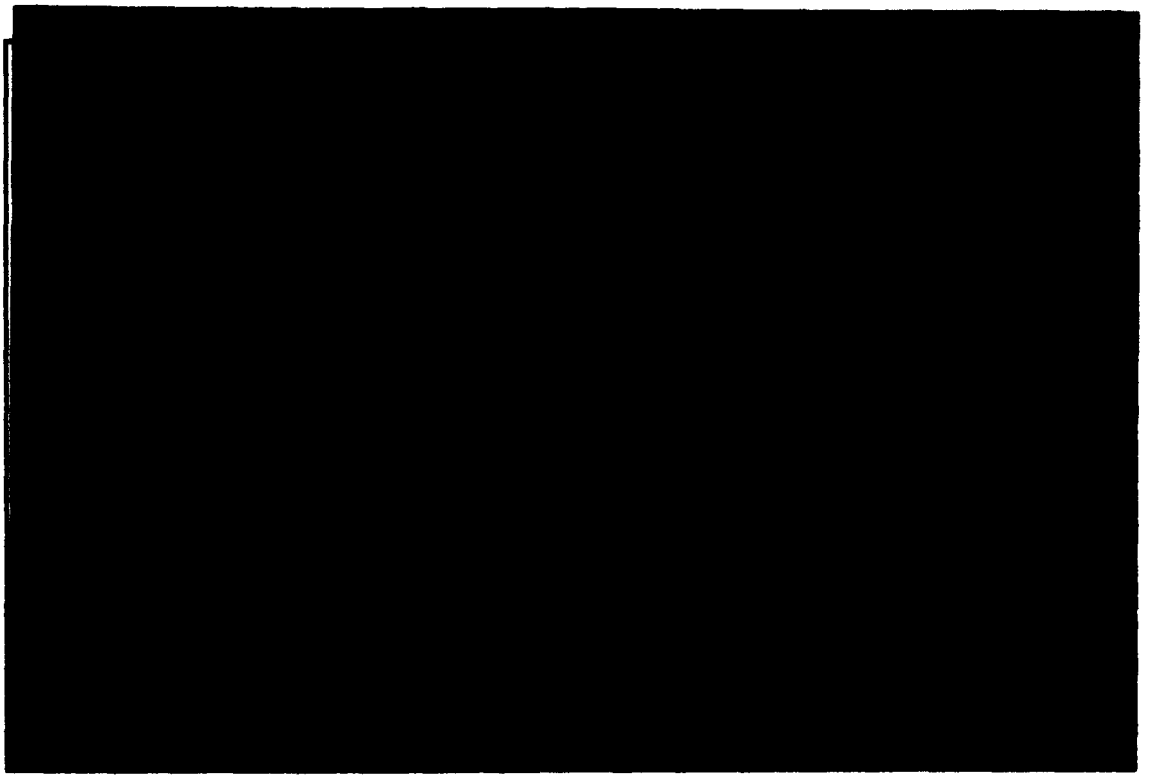


Figure 4.18: Capability Contour Map One

Figure 4.19: Capability Contour Map Two

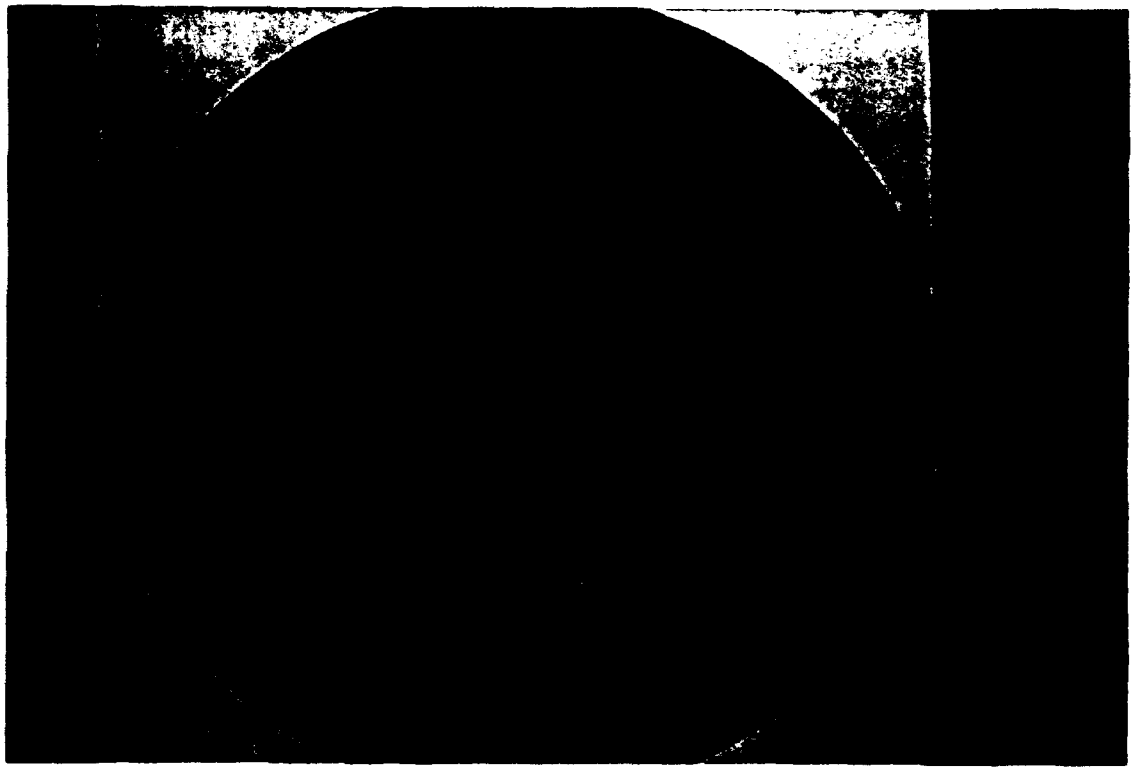


Figure 4.20: Capability Contour Map Three

4.2.7 FLS_Player Class

The FLS_Player Class provides the actual representation, placement, and manipulation of the Sentinel watchspaces within the simulation. All rendering issues, such as initialization, size, movement, geometry, and viewing, are taken care of by the FLS_Player Class. There are two main issues with handling the Sentinel watchspaces within the simulation: geometric representation and transparency issues.

The first issue on geometric representation deals with what geometric volume best characterizes a Sentinel watchspace. There is currently a choice between four geometric volumes that could apply to a Sentinel watchspace: irregular shaped volumes, cube, hemisphere, and cylinder. I address each of these in turn and discuss their strengths and weaknesses. Finally, I talk about why we chose the last one, the cylinder, as the geometric representation of a Sentinel watchspace.

- Irregular shaped volumes: while the use of an irregular shaped volume would allow the user total freedom on the shape of the Sentinel watchspace, it would be computationally prohibitive to determine the Sentinel watchspaces that contain specific entity players. Also we would have to have a different model for each Sentinel watchspace specifically defined.

- Cube: while the cube is just a simplified version of the irregular shaped volume, it would be much easier and less expensive to calculate the entities that are in specific Sentinel watchspaces. However, while the cube is probably the easiest of all the geometric volumes to do the contained-in calculations, it is not representative of the true spirit of the Sentinel. The Sentinel, in theory, is suppose to mimic a scout out in the field. With this in mind, the scout would not see the surrounding area as a cube, but he would see the surrounding area in a radial fashion. This idea leads to the next two geometric volumes.

- Hemisphere: the hemisphere seems like the perfect choice for representing a Sentinel watchspace. However, if high flying aircraft and missiles are important to the Sentinel, then these objects would not fall within the Sentinel watchspace representations unless they were big enough to accommodate this. The choice of a hemisphere is a good one if we could divide the Sentinel watchspaces into different types of Sentinel, i.e. a ground Sentinel and a air Sentinel. In this case, the hemisphere would be a perfect choice for a ground Sentinel. However, this current implementation of the Sentinel system does not make allowances for different types of Sentinels.

- Cylinder: the cylinder is the implementation choice we have made for the Sentinel. The cylinder can represent the radial viewing of a scout in the field, as well as, be able to include high flying planes and missiles. Containment calculations can be done without very much expense, although not as cheaply as a cube would be. Also, modifications to the cylinder shape allows us to represent a Sentinel watchspace with a cage

like volume representation. Figure 4.21 and Figure 4.22 shows both the regular cylinder representation for Sentinel watchspaces and the cage representation for Sentinel watchspaces respectively.

The second issue on transparency deals with implementing the Sentinel watchspaces in such a way so that all other objects and features within the simulation are viewable inside and through the transparent Sentinel watchspaces. The transparent Sentinel watchspaces do not have a problem with solid objects inside of their volumes or beyond their volumes. A problem occurs when we try to view other transparent objects or other transparent Sentinels through a transparent Sentinel. The z-buffer algorithm for Performer does not properly handle rendering transparencies in the scene. A simple fix to this problem is to render the transparent objects last ([Fol90:755]). However, we must also keep in mind that the Sentinel watchspaces themselves need to be rendered in back-to-front order from the current view point so that we can view Sentinel watchspaces through other Sentinel watchspaces.

Figure 4.23 and Figure 4.24 shows the Sentinel watchspaces in fly mode within the SBB. Notice how the Sentinel watchspaces appear darker when they are viewed through other Sentinel watchspaces. Also notice that in Figure 4.23 one of the Sentinel watchspaces appear to be highlighted in red. The FLS_Player Class does a model switch everytime the user attaches to a Sentinel watchspace. This model switch shows the attached Sentinel watchspace as highlighted to indicate to the user which watchspace they are attached to. Figure 4.24 shows a number of objects located within the scene. These objects have transparent locators around them so that they can be seen at great distances. However, since they are rendered before the Sentinel watchspace cylinders are, we can see them inside and behind the transparent Sentinel watchspaces. Once again, notice how the transparent objects appear darker when viewed through a transparent Sentinel watchspace cylinder.

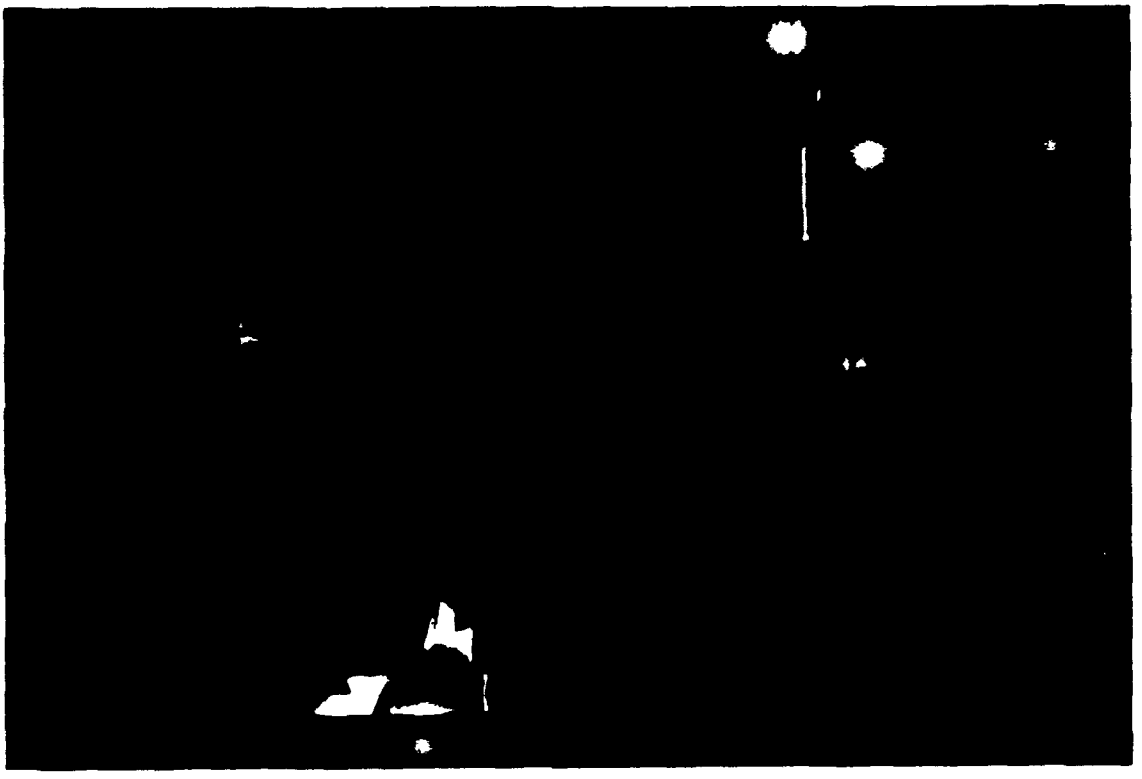


Figure 4.21: Sentinel Watchspace Cylinder Representation.

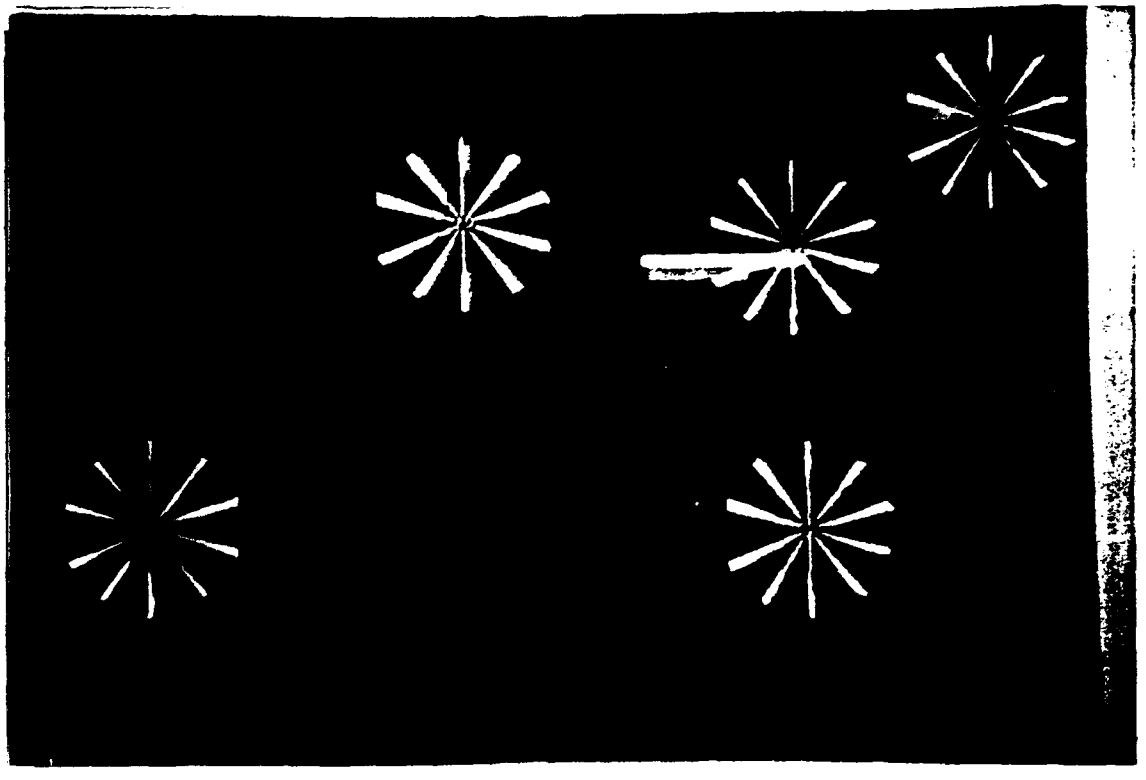


Figure 4.22: Sentinel Watchspace Cage Representation.

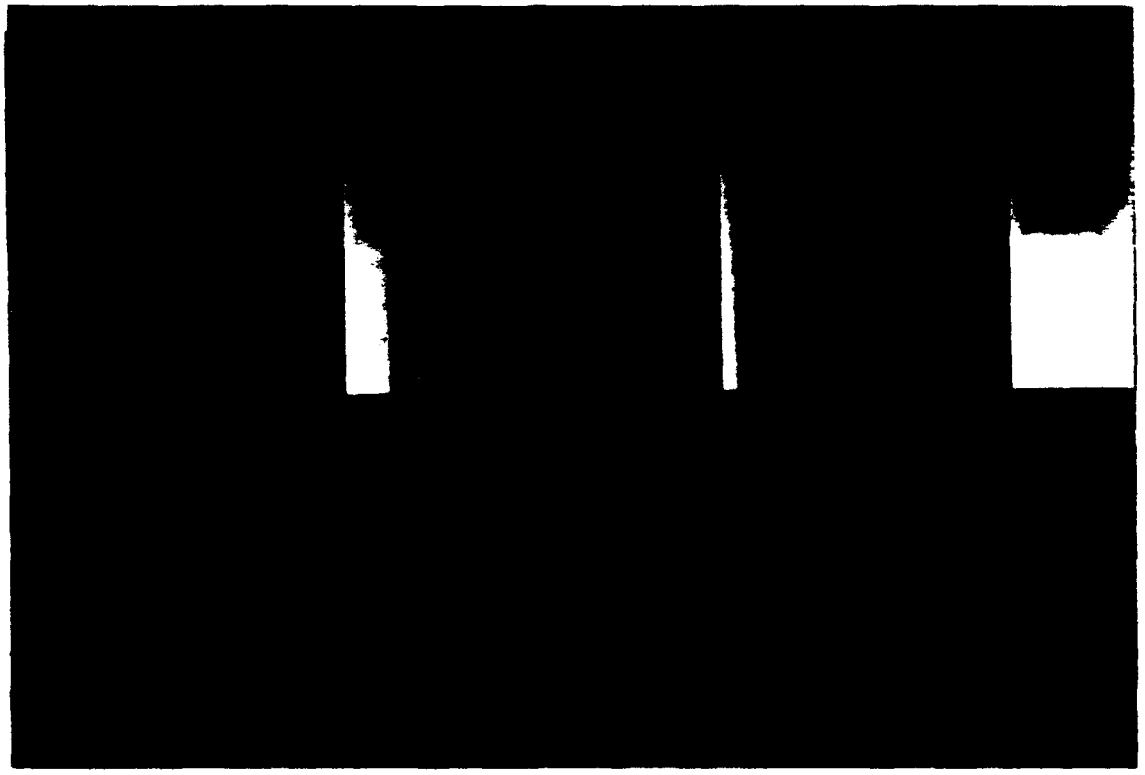


Figure 4.23: Transparent Sentinel Watchspaces in Fly Mode

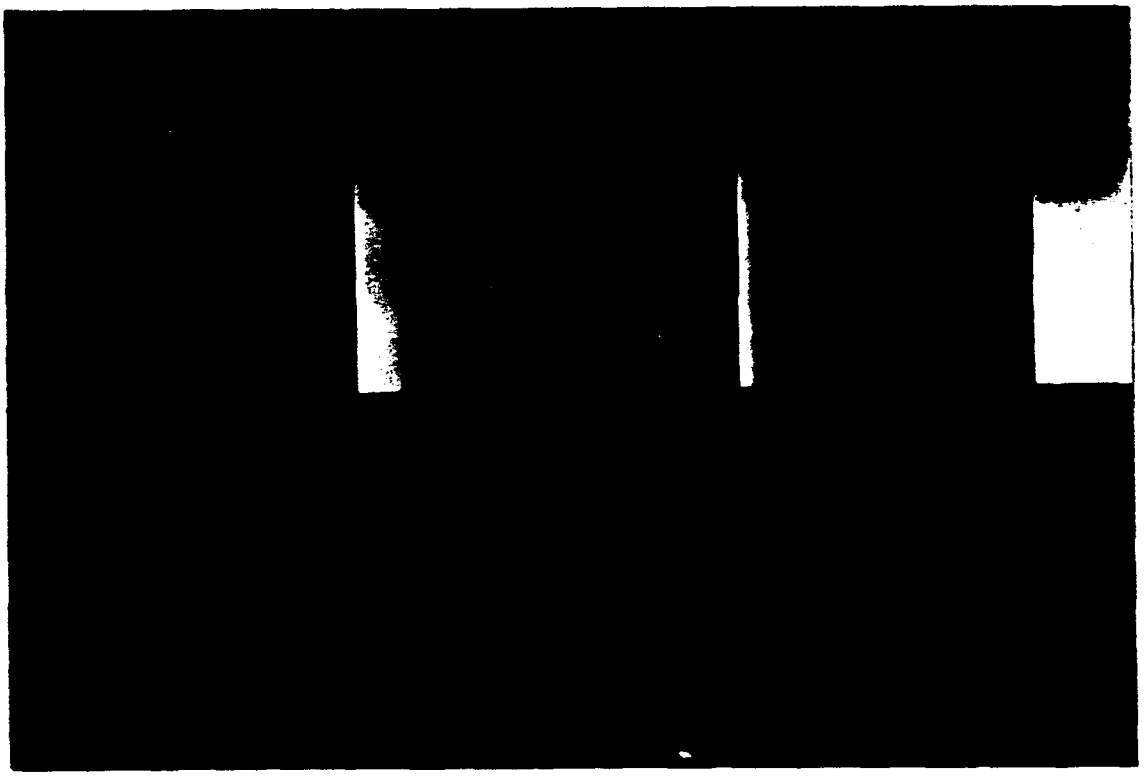


Figure 4.24: Transparent Sentinel Watchspaces With Active Players.

Although not shown, the cage representation allows us to use a non-transparent Sentinel watchspace within the scene. Because the spokes of the cage allow for easy viewing into and through the Sentinel watchspace, making the cage transparent is not necessary. However, while the cage representation works well in plan mode, it can be confusing to the user in fly mode.

4.3 System Integration

This section deals with the integration of the Sentinel system with various other programming libraries, frameworks, applications, and toolkits. The order of the subsections indicates the relative importance of each component to the Sentinel system.

4.3.1 Forms 2.1

The main library that the Sentinel system integrates to for user interface is the Forms 2.1 Library ([Ove92]). This library provides a graphical user interface toolkit for the Silicon Graphics Workstations. The main purpose of this library is to allow the user to develop graphical user interfaces that are easy to use and fast to develop ([Ove92: i]).

The main framework for each user control panel of the Sentinel system was created using a forms design application included with the forms library package. This application goes by the name of Forms Designer. Forms Designer created all the user interface control panels associated with the Sentinel system ([Ove92: 33 - 47]).

Integration of the forms library with the Sentinel system requires four parts or steps. The first is the inclusion of the actual code generated by the Forms Designer within the units and classes of the Sentinel system. The second is the linking of the actual forms library into the framework application (SBB). The third is the placement of the forms library initialization call, fl_init(), into the framework application (SBB). The last is the placement of a control loop that is responsible for checking all forms for user interaction.

The previous chapters on design and implementation discussed the first item. The second item requires the addition of the forms library to the makefile for the overall application. The next subsection on the integration of the Sentinel system with the SBB addresses the third and forth items.

4.3.2 Synthetic BattleBridge

The Synthetic BattleBridge (SBB), redesigned by Capt. Kirk Wilson, is the driving application that the current version of the Sentinel system integrates with ([Wil93]). The SBB takes care of assimilating all the simulation data and passing it to the Sentinel system for further processing. The SBB is the driving application, and the starting point for all the managers, frameworks, and other support tools needed by the SBB and Sentinel System. With this in mind, the SBB needs to pass information to the Sentinel system by means of method calls to the FL_Sentinel Class. The FL_Sentinel Class also derives its draw thread from the stealth draw thread located within the SBB framework.

The integration of the Sentinel system into the SBB consists of two parts. The first part is just the inclusion of the structured programming library units and the Sentinel system classes into the compiling and linking of the driving application. The second part deals with the placement of Sentinel system class method calls within the framework of the SBB application source code. The following paragraphs discuss these method calls at a very basic level. The programmer's manual in Appendix II talks about the exact use of these calls, and parameters needed by these methods.

The first part of the integration of the Sentinel system into the SBB requires that the driving application's makefile be modified to include the needed Sentinel libraries and classes. There are currently four libraries that need inclusion in the compilation of the driving application: libfls_comp_os*.a, libfls_config_os*.a, libfls_counts_shared_os*.a, and libfls_outputs_os*.a. The "*" in the library names indicate whether you want to compile on the Silicon Graphics operating system version 4.0.x or version 5.x. Just

replace the "*" with the appropriate number (4 or 5). The structured programming units these aforementioned libraries hold are the computational unit, the configuration unit, the input unit, and the output unit and control unit respectively. Note that the libfls_outputs_os*.a library contains both the output unit and control unit. There are currently two Sentinel system source code files that contain the Sentinel system classes that must also be included in the compilation of the driving application: FL_Sentinel_mgr.cc and FLS_player.cc. The classes contained in the aforementioned source code files are the FL_Sentinel Class and the FLS_Player Class respectively. Note that the appropriate header files for the classes must also be included.

The second part of the integration deals with the placement of Sentinel system method calls within the source code of the SBB. There are two SBB source code files that must be modified to integrate the Sentinel system: stealth.cc and sbb_app.cc.

In stealth.cc there are only two things that need to be done to integrate the Sentinel system. The first is to include the Sentinel class header files and create the appropriate Sentinel system variables needed. The second is to place the FL_Sentinel Class method call, draw(), into the draw thread of the stealth player class. In this way, the FL_Sentinel Class can now use the draw thread of Performer through ObjectSim, to place things on the screen.

In sbb_app.cc there are nine steps that need to be done to fully integrate the Sentinel system into the SBB.

As in stealth.cc, the first step is to include and declare the appropriate header files and Sentinel system variables.

The second step is to add a command line argument so that the Sentinel only comes up with the SBB when the user gives the corresponding command line argument. For the current application, a command line argument of -z invokes the Sentinel system upon startup.

The third step to do is to initialize the FLS_Player Class shared memory structure. This is done in the SBB main by using the FLS_Player Class method called init_shared().

The fourth step is to setup and initialize Sentinel system shared memory variables that control the overall geometric representation of the Sentinel players (cylinders). This allows the user to hide or change all of the geometric representation of the Sentinel player. This is done in SBB_App::initialize().

The fifth step to be done is to assign space to and initialize the Sentinel players (cylinders). This is done by using two Sentinel system method calls in a row: config() from the FL_Sentinel Class and assign_space_and_initialize_FLS_players() from the FLS_Players Class.

The next four steps required to integrate the Sentinel system into the SBB are placed in procedure calls that use the ObjectSim framework: SBB_App::init_sim, SBB_App::init_draw_thread, SBB_App::pre_draw, and SBB_App::propagate.

In the SBB_App::init_sim we orient the Sentinel player's initial view in the simulation. This is done with the FLS_Player Class method call init_sim_FLS_players().

In the SBB_App::init_draw_thread we initialize the Sentinel system with the current entity state of the simulation from the Object Manger (described in subsection 4.3.4). Also, we start up the user interface control panels via the forms library. This is done by two method calls: init() and start_forms() both from the FL_Sentinel Class. Lastly, we place the initialization call for the forms library (fl_init()) here before we use the start_forms() method mentioned above.

In the SBB_App::pre_draw we place the method that updates the Sentinel system with the current entity state of the simulation. This is done with the FL_Sentinel Class method update(). We also check to see if we need to reset the cursor back to normal mode with the FL_Sentinel Class method reset_default_cursor(). It is done here because redrawing the cursor has to be done on the draw thread. Also, we set up the use of the F11

key with the Sentinel system to control representation of the Sentinel players. As mentioned earlier, the user can toggle the Sentinel players on or off, or change their geometric representation if setup to do so. Lastly, we place the forms library call, fl_check_forms(), in the SBB device queue loop so that the Sentinel system can monitor interactions with the user control panels developed with forms 2.1.

Finally, in the SBB_App::propagate we use a method call that takes in all information on the current state of the Sentinel players and makes the appropriate changes as needed to reflect these changes in the scene. This is done with the FLS_Player Class method propagate_FLS_players().

As noted earlier, the method calls shown do not indicate input or output parameters. The programmers manual in Appendix II gives the exact use of these method calls.

4.3.3 ObjectSim and Performer

The rendering process used by the driving application (SBB) is a application framework created by Capt. Mark Snyder. This framework, known as ObjectSim, is the go between for the driving application and the graphics library, known as Performer, for the Iris and Onyx workstations ([Sny93]). Currently, the Sentinel system can work with Performer 1.2.

While the driving application (SBB) takes advantage of most of the capabilities of the ObjectSim application framework, the Sentinel system uses two aspects of ObjectSim to render the Sentinel players on the screen. The first use of ObjectSim allows the Sentinel system to setup and use the draw and application threads. The Sentinel system can then use these threads to render Sentinel players in the scene. The second use of the ObjectSim application framework by the Sentinel system takes advantage of the Performer rendering tree setup by ObjectSim. By placing the Sentinel players in the correct spot in the Performer rendering tree, ObjectSim can guarantee that the Sentinel players render last.

This is obviously very important based on earlier discussions dealing with transparent object rendering in the scene.

Figure 4.25 shows the Performer rendering tree associated with the ObjectSim application framework for the SBB. The figure also shows the location of the Sentinel players in the tree. Since Performer traverses the rendering tree in a depth-first order, the tree traversal is from top to bottom and left to right ([Mcl92: 5-1 - 5-2]). Therefore, as shown in Figure 4.25, this causes the Sentinel players to be rendered last.

To use both ObjectSim and Performer 1.2 we need to include their appropriate libraries and header files while compiling and linking the driving application (SBB). The programmers manual in Appendix II gives the particulars on how this is done.

4.3.4 Object Manager

The Object Manager, a continuing research effort by Mr. Steven Sheasby, is the interface between the network and the driving application (SBB) ([She92]). The Object Manager interprets the DIS 2.0 protocols coming over the network and sets up the appropriate data structures in shared memory on the current entity state of the simulation. Once again, the driving application (SBB) sets up and uses most of the capability of the Object Manager. However, the Sentinel system needs to do two things in order to integrate the Object Manager with the Sentinel system.

The first thing needed by the Sentinel system from the Object Manager is the instantiation of the Object Manager from the driving application (SBB). The Sentinel system needs this so that it can pull information off of the shared memory structures dealing with the current entity state of the simulation.

The second thing needed by the Sentinel system from the Object Manager is a listing of all the needed entity types and designations from the DIS 2.0 protocol.

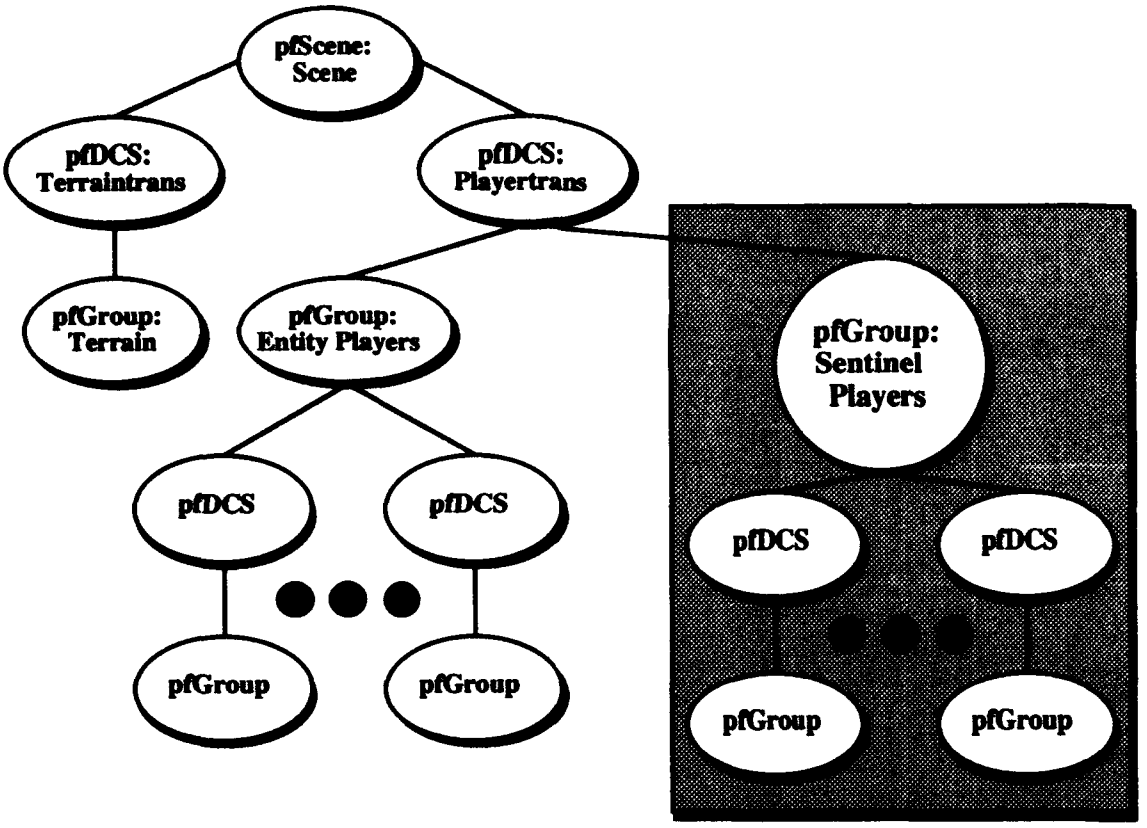


Figure 4.25: ObjectSim Created Performer Rendering Tree.

The Sentinel system uses this information to set up array structures that can keep track of entity types and designations for each Sentinel watchspace. Subsection 4.4.2 on the DIS Entity Manager utility explains how the Sentinel system acquires and sets up the entity type and designation information from the Object Manager enumerated type files.

4.3.5 Model Manager

The Model Manager was created by Capt. Kirk Wilson. It is a utility used to help manage all the different models and terrain files needed by any driving application ([Wil93]). The Sentinel system uses the Model Manager instantiated by the SBB to manage the different geometric representations that a Sentinel player can use. By doing this, all the Sentinel system needs to do to add a new geometric representation for a

Sentinel player is to add a new index number associated with the new geometric representation to the appropriate Model Manage data file. The user manual in Appendix I, and the programmers manual in Appendix II gives the particulars on how this is done.

4.3.6 Sound Generation Facility

The Sound Generation Facility, created by Capt. Chuck Wright and Capt. Brian B. Soltz, allows any driving application to add sound to the application. The SGF is a simple, maintainable monaural and stereo sound generator for any Virtual Reality System (VRS). The platform for the VRS is a Silicon Graphics Iris or Onyx Workstation and the platform for the SGF is an Apple® Macintosh™ IIci or Quadra 800. The two systems are directly connected via an RS-232 serial port. Requests for sounds are issued by the VRS via the Mac Sounds Class, and the SGF responds by playing the requested sound. Since there is no network traffic or communications overhead, sound requests are played in near real time ([Sol92]).

The Sentinel system currently uses the SGF to play messages about possible interrupt conditions occurring within a Sentinel watchspace. These sound messages are another cueing mechanism used by the Sentinel system to help with situational awareness assistance for the user.

4.4 System Utilities

While the design of the Sentinel system gives the user a drop in type module that extends the driving application without the use of any other tools, two Sentinel system utility tools were also created to help manage, create, and modify some of the configuration files needed by the Sentinel system. The following subsections briefly address the purpose and use of these two utility tools. The first utility helps with the initial placement of Sentinel watchspaces and entity weights, while the second utility manages the needed entity types and designations required by the Sentinel system.

4.4.1 Sentinel Watchspace Configurer

The Sentinel Watchspace Configurer (see Figure 4.26) allows the user to create and modify watchspace definition and entity weight files without having to run the driving application that supports the Sentinel system. In this way, the user can create new or different default files to match different simulations or scenarios. Then, before running the driving application, all the user needs to do is change the name of the file they want loaded at run time to the file name for the default that is loaded at runtime. Note that this is only necessary with the default areas file and not the default entity weight file. This is because the user can load in other entity weight files during the running of the programming by invoking the "Config Sentinel" icon button. However, there currently does not exist a facility to allow the user to load watchspace definitions on the fly.

To use the Sentinel Watchspace Configurer, launch the program named config_FLS_SA. Then press the "load" button to load in files to work from or start from scratch. The entity weight editor works exactly the same as the "Config Sentinel" facility found with the Sentinel system. To enter Sentinel watchspace definition information, follow the instructions provided on the screen next to the watchspace definition input area. Figure 4.26 shows the startup screen for the Sentinel Watchspace Configurer.

4.4.2 DIS Entity Manager

The DIS Entity Manager allows the user to specify what entity type and designations the Sentinel system should keep track of. It does this by converting a user configuration file of entity types and designations into a header file which defines index numbers for each entity type or designation. We then compile with this new header file so that the Sentinel system can use these index numbers to keep track of the entity types and designations. A default entity weight file is also created based on the entity types configuration file with each of the entity types given a default weight of 1.0.

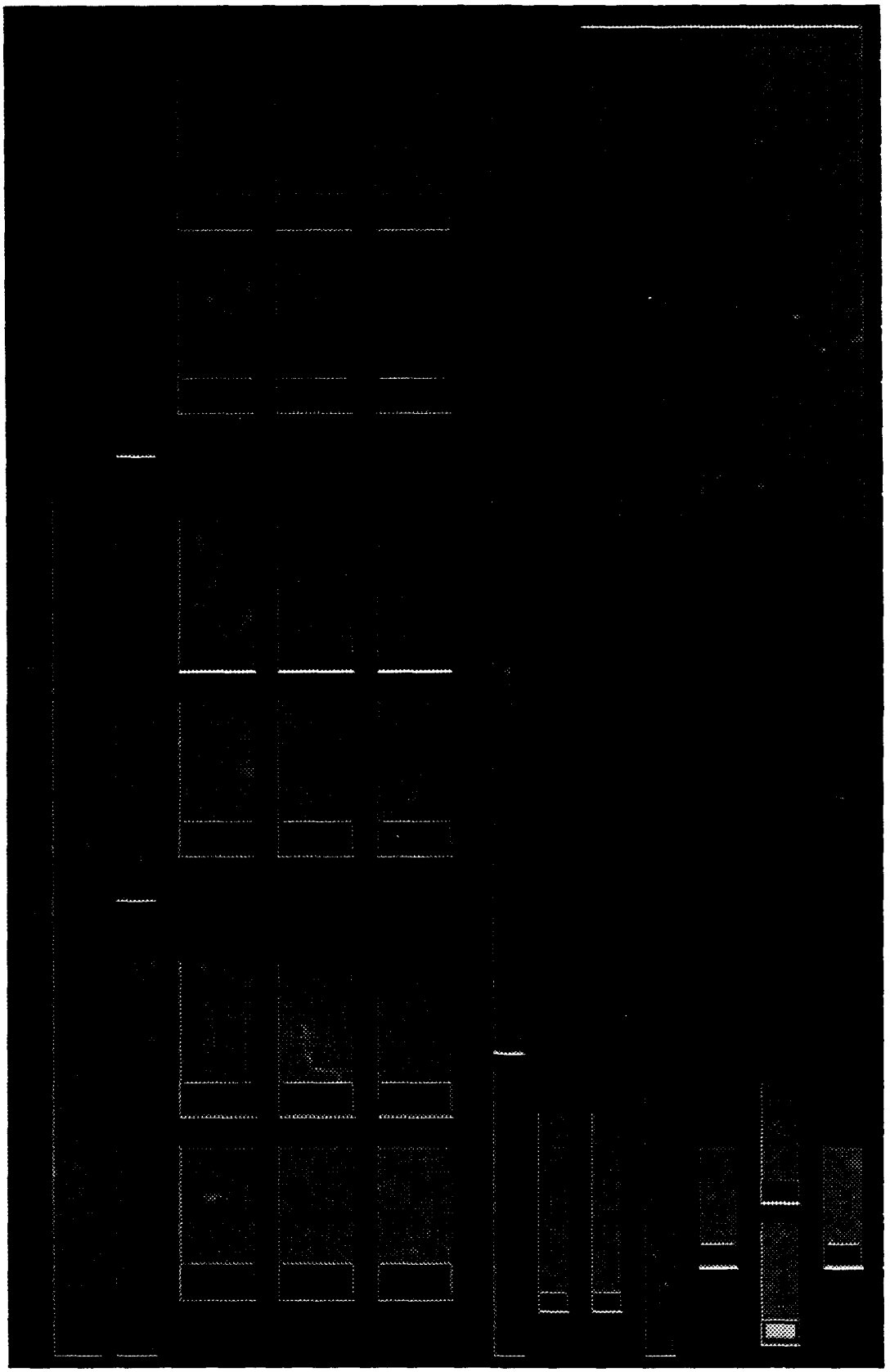


Figure 4.26: Sentinel Watchspace Configurer

The only restriction is that the entity type configuration file can only have names of entity types and designations that match completely with the enumeration names given in the Object Manager source code. The entity type configuration file should also have a name of Object_Types.dat. From this file, two files are created with the names FLS_Object_Types.h and default_obj_weights.dat. Theses files contain the defines and entity weight information respectively.

4.5 System Operations

Upon startup with the -z command line argument, the Sentinel system reads in all configuration information and initializes the user control panels and fuzzy logic computation unit. All Sentinel watchspaces are initialized and placed in the simulation. Once all the initialization is completed, the Icon level control panels appear on the screen. Note that the Sentinel system performs two operations over and over again until the driver application is shutdown.

The first operation causes the Sentinel system, at periodic intervals, to retrieve current entity state information about the simulation. This information is then processed with the fuzzy logic computation unit. The results are then fed back to the output unit and displayed to the user on the screen.

The second operation is more of a control panel navigational task. As mentioned in subsection 4.2.3 on the implementation of the control unit, there are a number of user control panels put up on the screen depending upon what the user is currently trying to do. The Sentinel system keeps track of the user state based upon the control panels that are open, as well as, the control panel buttons that the user pressed. Figure 4.27 shows a navigation through all the control panels and their buttons.

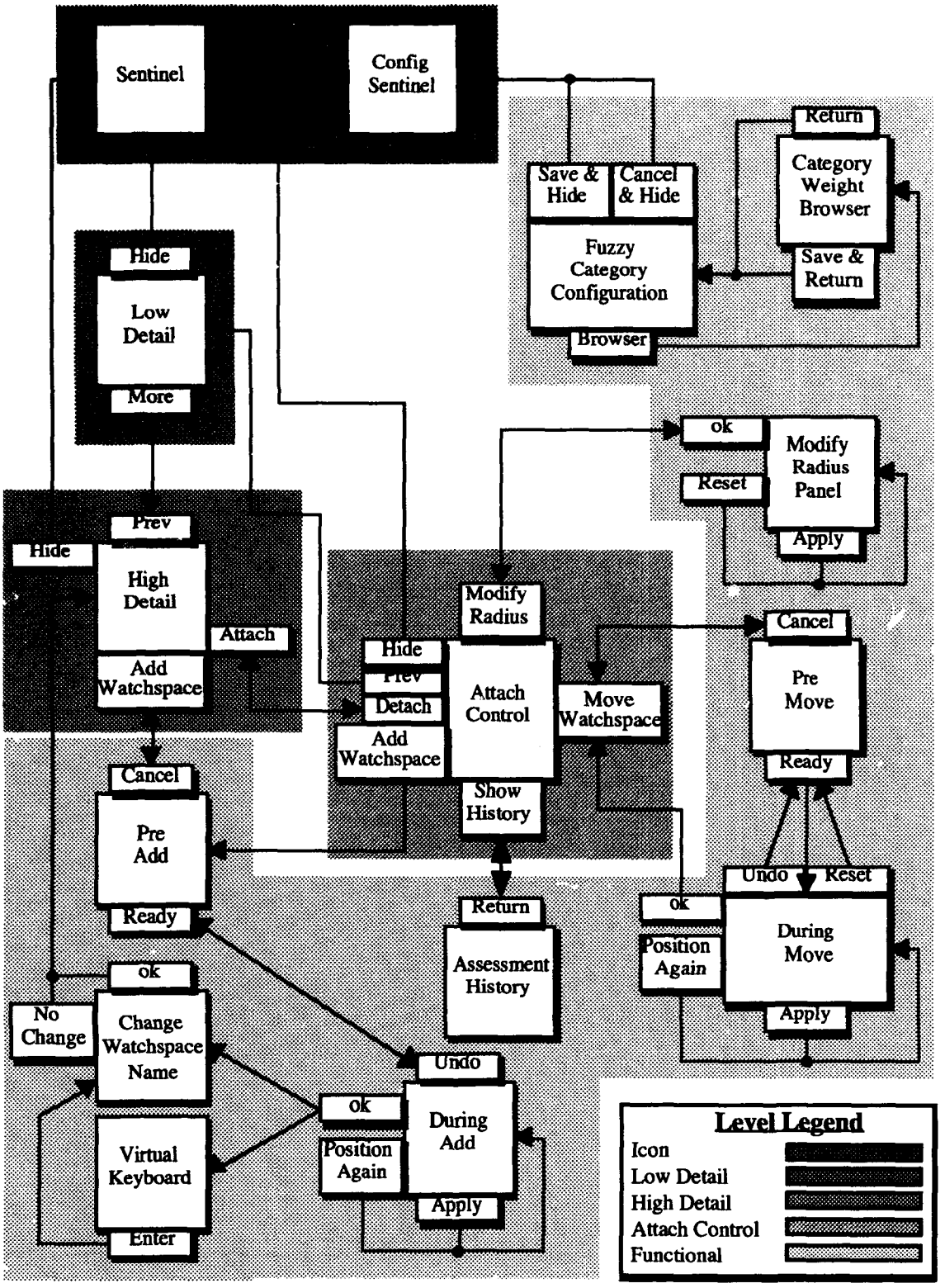


Figure 4.27: User Control Panel Navigation

4.6 Conclusions

The implementation of the Sentinel system with the SBB incorporates a drop in module approach. The idea being that with minimal changes to the source code of the driving application, the programmer can extend the driving application to include the Sentinel system. The implementation with the SBB was very successful from, this point of view. There was minimal source code changes need in the SBB and the class and library structure made it easy to compile and link in the Sentinel system with the SBB application.

The next chapter discusses some results and recommendations for this thesis project and future research respectively.

V. Results and Recommendations

5.1 Introduction

This chapter addresses the results of implementing the Sentinel system with a synthetic virtual environment in a large scale battlespace. For this particular implementation, we used the Synthetic BattleBridge (SBB) to represent the large scale synthetic virtual environment. The rest of this chapter presents with some observations about the use and integration of the Sentinel system along with problems encountered during the effort, as well as, possible future research and development in this area.

5.2 Observations

The Sentinel system, as tested with the SBB, is a capable situational awareness tool. However, due to the restricted ability to test the Sentinel system against great numbers of different entity types participating in a simulation, true results on how the Sentinel system behaves is limited. In other words, unless the Sentinel system is stressed at its maximum numbers, we can not determine if the watchspace assessments produced match closely to what we would expect given the same information presented to a battlefield commander. Numbers in the range of 50 to 500 for each fuzzy logic category are required to assess the Sentinel system's correctness. However, this would mean that the driving application would have to support 900 to 9000 objects in the simulation, as well as, the Sentinel system processing that same number of objects for possibly each Sentinel watchspace. Also, it would not be enough to just broadcast 900 to 9000 objects randomly. To get meaningful test results, these objects would have to be involved in specific scenarios to trigger the rules we want to test in the rule base for the fuzzy logic computation unit.

5.3 Problems Experienced

There were two main problem areas encountered with the design and implementation of the Sentinel system with the SBB. The first of these areas was how to design and create a user interface that was easy to use. The second problem area deals with the overhead of the Sentinel system on the graphic pipeline that results in a lower frame rate.

The first area was a trade off between ease of use for decreased frame rate, and faster development time for a less complicated design and implementation. In other words, the first trade off gives us user friendliness but at a slower frame rate, while the second trade off allowed for fast development in a less complicated way. The forms library was chosen with the above trade offs in mind.

The forms library is a programming shell that allows a programmer to make less complicated graphic calls to produce very complex graphic objects. However, since the forms library is just a shell over the GL calls of the Silicon Graphics Workstation, there is overhead involved which leads to a slower frame rate in some instances. This slower frame rate depends on many factors. How often we write to the forms, where the forms are located in reference to the rest of the rendered scene, and how much user interaction is needed for each form are just some of the more important factors associated with the change in frame rate.

However, what the forms library costs us in speed, it more then makes up for in easy of use and faster development time. The control panels created with forms are very pleasing to look at and simple to use. However, where the forms library really comes through is in the design and implementation of the control panels themselves. With the use of the Forms Designer application that comes with the forms library package, development time was approximately one quarter of the time it would have taken to develop the control

panels using straight GL calls. Also, changing an existing forms control panel can be done in a matter minutes using the Forms Designer application.

5.4 Future Research and Development

Based on this current implementation, we have concluded that the fuzzy logic paradigm has the potential to provide the type of situational awareness assistance needed in a virtual battlespace environment. However, several aspects of the implementation need improvement. First, the network of nodes and variables used in our initial implementation must be expanded, along with the inferences required to reach subgoals within the semantic network. This will require a better characterization of the phase of a battle and of different battlespace situations. For example, a given situation early in the battle may have a different level of threat than the same situation later on in the battle. We must also develop a better characterization of the interrelationships and differences between air and land combat. It will also be necessary to expand the relative fire power ratings, i.e. rank order, of each model of weapon used in the simulation and to match these entities to the groups to which they belong. This assignment and matching process will be based on the given information provided by Army FM 100-2-1, 100-2-2, and 100-2-3, and on interviews with commanders.

Because the virtual battlespace environment is growing in complexity from the current 100 objects to over 8000 total objects, we will have to implement a capability to resolve operator feedback conflicts between Sentinels. This may be accomplished through the use of fuzzy modifiers to give us additional levels of commander notification. Another change is being forced upon us by the growth in complexity of the simulated battlespace. Currently, the Sentinels only analyze volumes and situations that correspond to the volume of concern for a battalion commander. Since the battlespace's volume is growing and becoming more complex, the Sentinel's capabilities must be expanded to support evaluations for commanders of larger combat formations.

In initial user studies, it was determined that the technique used to compute ω_{x_i} sometimes conflicts with commander's intuition about the situation in a Sentinel's watchspace. The Sentinel currently under development will compute ω_{x_i} in a manner that more closely resembles that of a tactical operations center.

Analysis has also indicated that the Sentinel should separate the computation of level of interest for ground forces from that for air forces. In other words, create a Ground Sentinel and a Air Sentinel. Also, the ability for Sentinel watchspaces to create themselves based on a certain set of rules or priorities would mean that the commander would not have to be perfect in the initial placement of the Sentinel watchspaces. Instead, as a watchspace triggers, the commander would be informed of the new watchspace and its placement. Further work must also be done in determining the most effective components of the rules as they relate to current military doctrine. For example, the types of objects and their relative firepower ratings must be as accurate as possible so that they reflect the military training philosophy inherent in the training of commanders, and so that the user studies can be effectively performed. Follow-on work will involve the recognition of troop/vehicle formations and the speed of these formations, the incorporation of elapsed time, orientation of weapons systems, and obstacles.

Supporting workstations need to be added to the standard configuration of the Synthetic BattleBridge. They will make the commander's situation within the virtual environment more like the environment the commander experiences in the real world. The supporting workstations will be used to provide entry points for staff members into the commander's station so that the staff can provide information and analysis to the commander without forcing him to leave the virtual environment.

One further area of future interest is the human factors of the display, particularly regarding the range and types of colors to be used, the design of the sliding scale, and the effects on the user of both the sliding scale and the level of interrupt.

5.5 Conclusion

The Sentinel system is a proof of concept for a graphical and computational tool that gives the user assistance in making decisions based on situational awareness information in the given virtual environment. The Sentinel system provides the user with a user friendly interface for easier operations and analysis of that current situational information in the simulation. The Sentinel system uses the powerful fuzzy logic set theory concept to closely mimic human decision making behavior. This fuzzy logic set theory concept allows the Sentinel system to combine and abstract many factors concerning the simulation and the chosen Sentinel watchspaces. The user then sees the results of this combination and abstraction as a relative color and bar length presented on the screen. As long as the user is able to determine the meanings associated with the color and bar length, the user can quickly determine the activity level in each Sentinel watchspace as compared to other Sentinel watchspaces. Therefore, through the use of the Sentinel the user can drastically cut down on the time needed to analyze and assess large amounts of information within the simulation. Analysis is now presented to them in a timely and efficient manner concerning watchspaces of interest which may not even be visible to the user in the current view of the simulation. This greatly enhances the situational awareness capabilities of any user.

APPENDIX I: USERS MANUAL

1. Overview

This user manual is provided to give a fast and easy tutorial on how to use the Sentinel system within the Synthetic BattleBridge (SBB). Section 2 discuses how to use the modifiable configuration files to setup initial conditions for the Sentinel system. Section 3 discusses how to actually run the Sentinel system within the SBB from startup to shutdown.

2. User Modifiable Configuration Files

The following subsections talks about how to modify the various configuration files in order to setup an initial state for the Sentinel system.

2.1 Object_Types.dat

The Object_Types.dat file contains a list of all the DIS entity types required and the category they belong in as designated by the user for the current simulation run. This enumerated list supplies various array structures with easy access to their array elements for processing. If new entity types need to be added to the simulation, all that needs to be done is to put the entity type name in the correct category as found in the Object_Types.dat file. However, this new entity type name must match exactly the enumeration name given to it in the Object Manager. Also, the Object_Types.dat file is used to build two other files for use in the Sentinel system: FLS_Object_Types.h and default_obj_weights.dat. Theses files contain the entity type defines and entity weight information respectively.

2.2 default_areas.dat

This file contains the initial locations of the Sentinel players within the simulation. It also contains information on what DIS representation is to be used: round (DIS_RE) or flat earth (DIS_FE). It also tells the system how many Sentinel watchspaces are to be setup

initially. Lastly, it tells the system the upper bound on the Fuzzy Logic Categories to be used when processing entity count information. The layout of the file is as follows: DIS_FE or DIS_RE, 16 floating point numbers indicating the Fuzzy Logic Categories' upper bounds, the number of initial Sentinel watchspaces, the lat-long position for each Sentinel watchspace along with a radius in miles and a name for the watchspace (8 characters maximum with no spaces). Example file that sets up four watchspaces in flat earth mode with varying Fuzzy Logic Category upper bounds:

```
DIS_FE
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0
4
30 21 23 N 60 12 09 E 1 Watchsp1
10 21 23 S 30 12 09 W 1 Watchsp2
15 21 23 N 45 12 09 E 1 Watchsp3
22 21 23 S 15 12 09 W 1 Watchsp4
```

2.3 xyz_FE_default_areas.dat

The system can also use an absolute x-y-z position for the Sentinel watchspace from a configuration file called xyz_FE_default_areas.dat. This allows for the direct input of x-y-z position information as opposed to lat-long position information gotten from the default_areas.dat file. In this case, the system ignores interpolation and uses the x-y-z position as absolute position relative to the local origin. The format for this file is as follows: number of initial watchspaces, then for each watchspace give the x, y, z, radius in meters, and name of the watchspace. Example file that sets up four watchspaces:

```
4
 5974.59     7175.41    -400.0     1609.3     Island
17516.64     6995.31    -400.0     1609.3     Town
17334.56    12173.65    -400.0     1609.3     Airport
20813.45    14423.78    -400.0     1609.3     Ranch
```

3. Running With the SBB

3.1 Startup

To startup the Sentinel with the SBB, all that needs to be done is to use the -z command line argument when starting up the SBB application.

3.2 Control Panel Navigation

The use of the Sentinel system is very straight forward. The Sentinel's user interface consists of a number of user control panels. The Sentinel system keep track of where the user is in relation to these control panels by the current state of the Sentinel system and what user input buttons are pressed. The following subsections present a quick journey through all the control panels and their functions. Each control panel is shown here for easy reference while reading. The starting point is the Icon Level of control panels shown in the lower left hand corner of the screen (see Figure I.1).

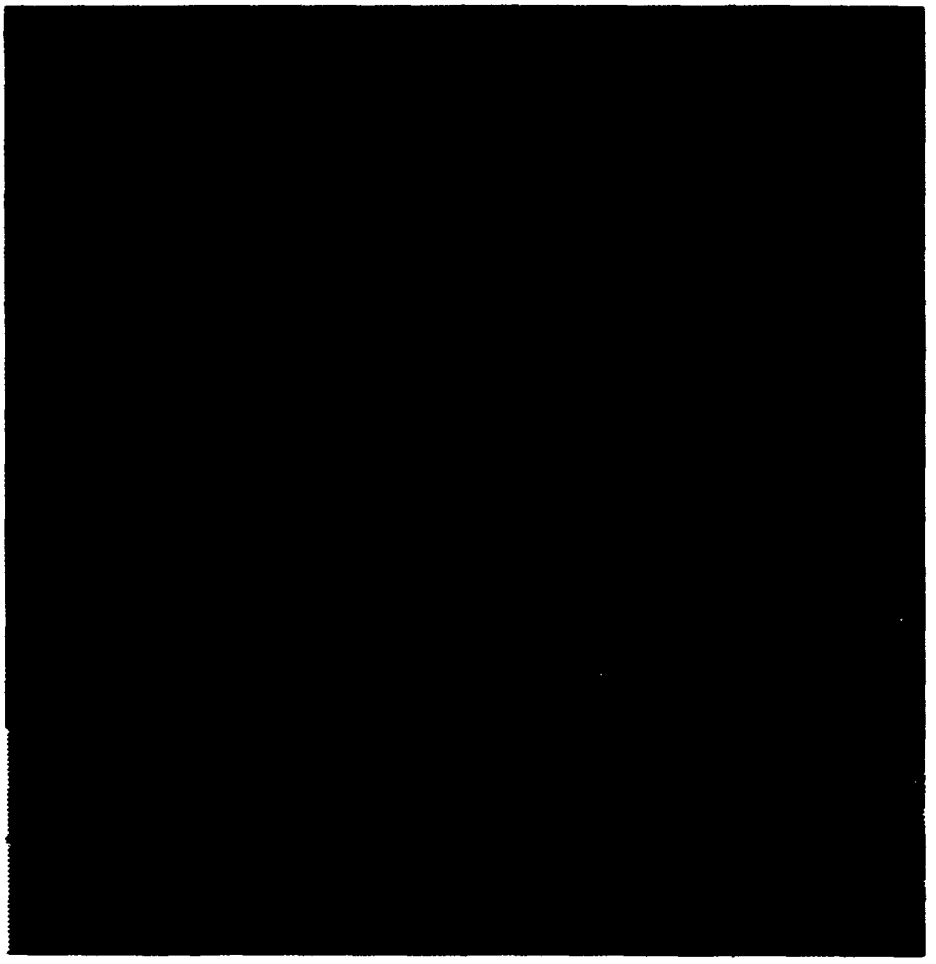


Figure I.1: Startup Icon Control Buttons

3.2.1 Config Sentinel

By pressing the Config Sentinel button shown in Figure I.1, the user is presented with a control panel that allows them to make changes to the entity weight for each type or designation of player in the simulation (see Figure I.2). The control panel itself is partitioned up into three functional areas: entity weight editing, DIS protocol viewing, and control.

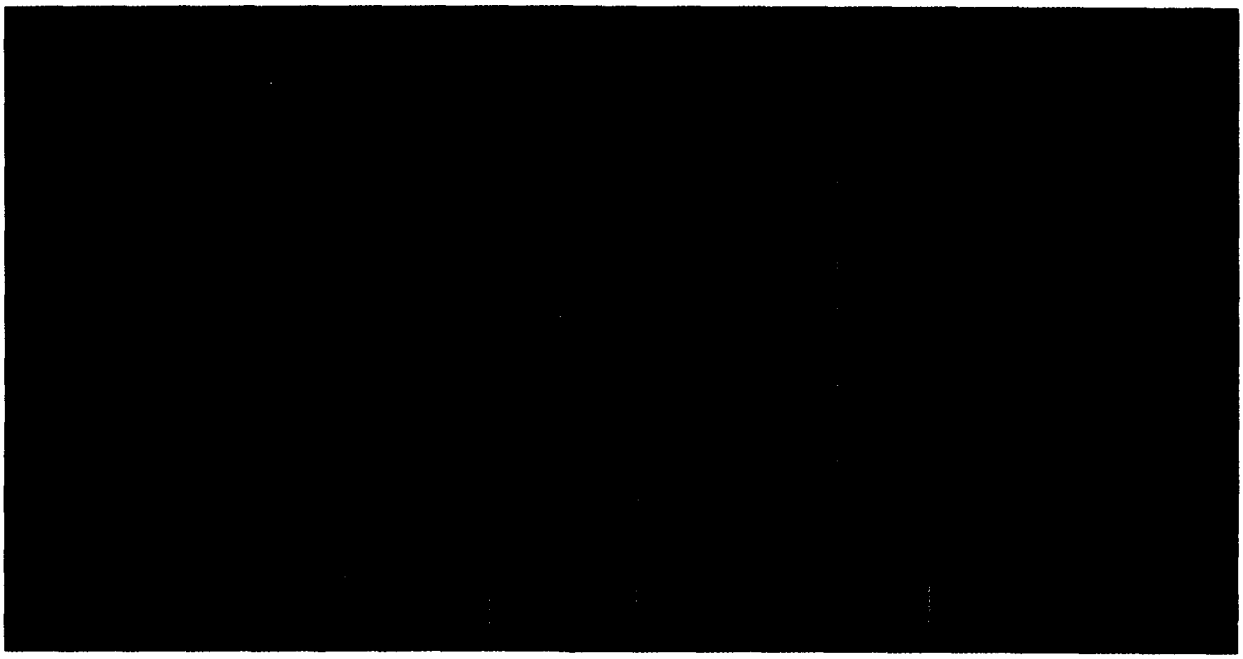


Figure I.2: Fuzzy Category Configuration Control Panel

When a user presses one of the 18 sub-object buttons, they are presented with a entity weight browser that allows them to go in and make changes to the entity weights for any type object in that particular category (see Figure I.3). When changes are made to the entity weights, the button for that category remains lit until it is saved or canceled (see Figure I.2).

The control area allows the user a number of options when using this control panel. While the DIS protocol area shows the user what DIS representation they have in the current simulation. This is for viewing purposes only, and the DIS representation cannot

be changed here. Currently, you must start SBB over again with changes made to the default_areas.dat file as mentioned in section 2.2 of this Appendix.

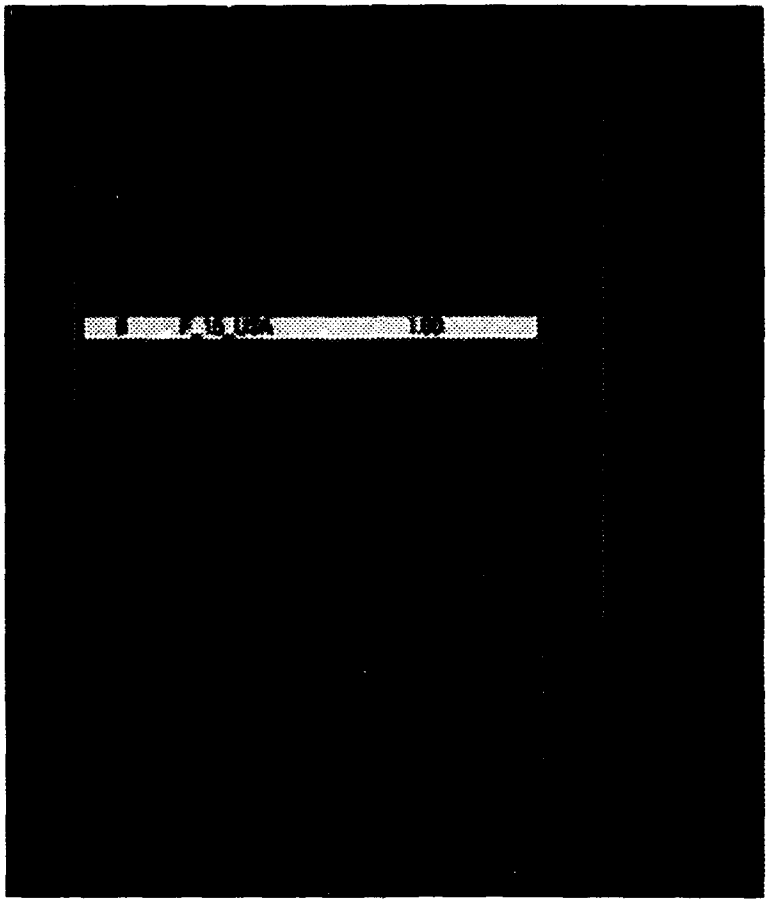


Figure I.3: Entity Weight Browser Control Panel.

3.2.2 Sentinel

The main control path for the Sentinel system comes from pressing the Sentinel button as shown in the lower left hand corner of Figure I.1. By pressing this button, the user is presented with greater and greater levels of control over the Sentinel system. The first level of control presented to the user when they press the Sentinel button is called the Low Detail level control panel.

3.2.2.1 Low Detail Level Control Panel

The Low Detail level control panel is merely a small viewer for watchspace assessments. It is presented to the user in the lower left hand corner of the screen, and provides minimal information about each currently active Sentinel watchspace (see Figure I.4). The only functionally allow to the user at this point is to attach or detach from a Sentinel player or to use the Hide and More buttons to go back to the Icon level or up to the High Detail level respectively. This control panel is really a go between from seeing no information on the Sentinel watchspaces to being able to control all aspects of the Sentinel watchspaces.

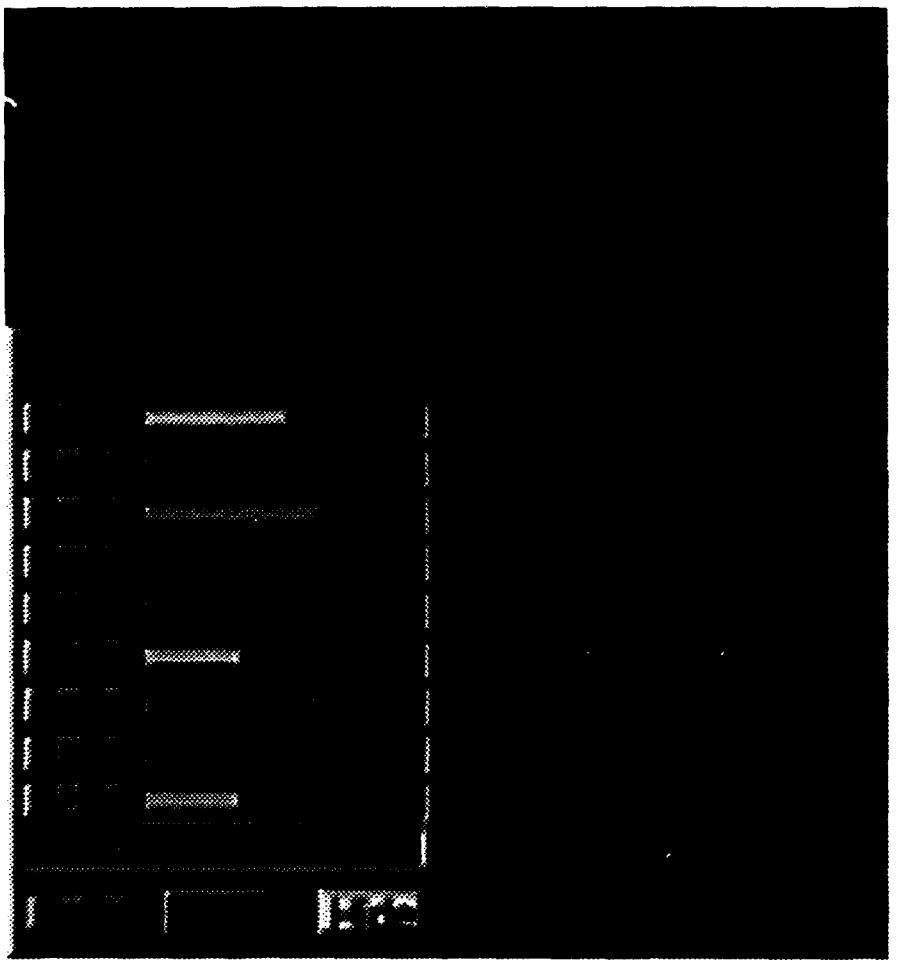


Figure I.4: Low Detail Level Control Panel.

3.2.2.2 High Detail Level Control Panel

The user gets to the High Detail level control panel by pressing the More button located on the Low Detail level control panel (see Figure I.4). It is at this control panel that the user can now begin to interact with the Sentinel players in the simulation. This control panel is not that much different from the Low Detail level control panel. The main difference is that now when you attach to a Sentinel watchspace at this level, other functions become available to the user. Also, at this level the user has the ability to add new Sentinel watchspaces to the simulation with the Add button (see Figure I.5). This level also gives the user more detailed information about the Sentinel watchspaces. The following subsections talk about adding and attaching to Sentinel watchspaces respectively.

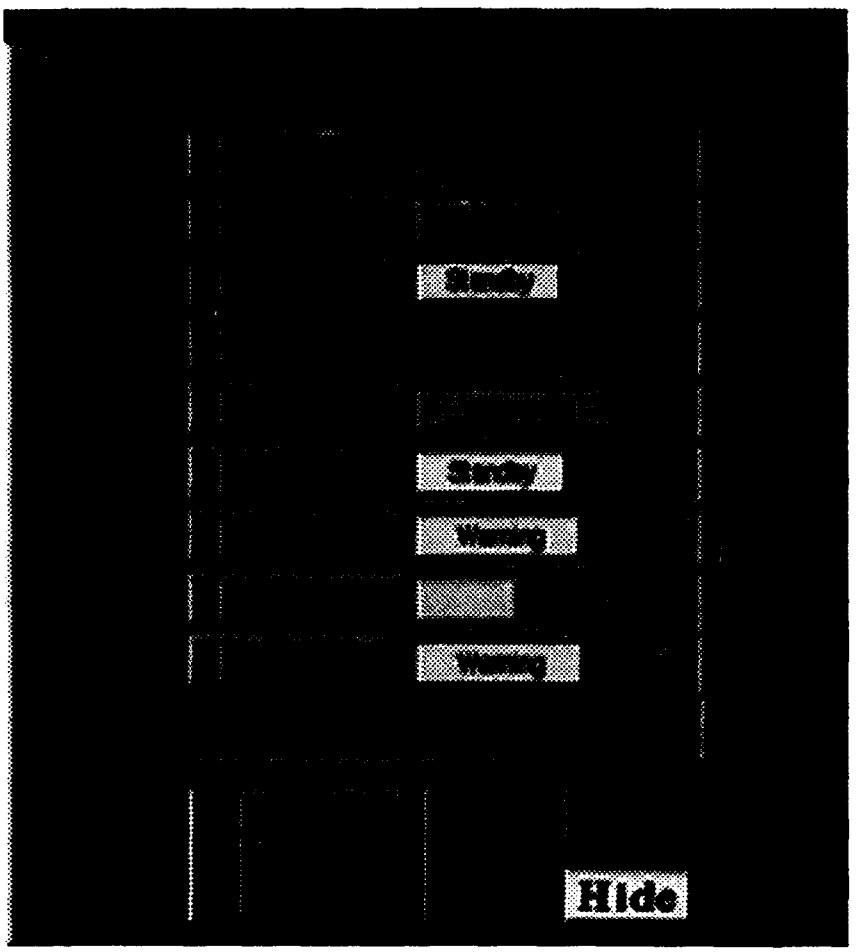


Figure I.5: High Detail Level Control Panel.

3.2.2.2.1 Add Watchspace Control Panels

When the user presses the Add button on the High Detail level control panel, the user is prompted through four types of control panels in order to place a new Sentinel watchspace in the simulation.

First the user is shown a pre-add control panel which instructs the user on how to proceed (see Figure I.6). When the user is ready to continue, the next control panel is presented, the during-add control panel (see Figure I.7). Lastly, when the user has successfully added a new Sentinel watchspace to the simulation, two more control panels are presented to the user: name change control panel and the virtual keyboard control panel. Both of these control panels allow the user to assign a name to the new Sentinel watchspace that was just added.

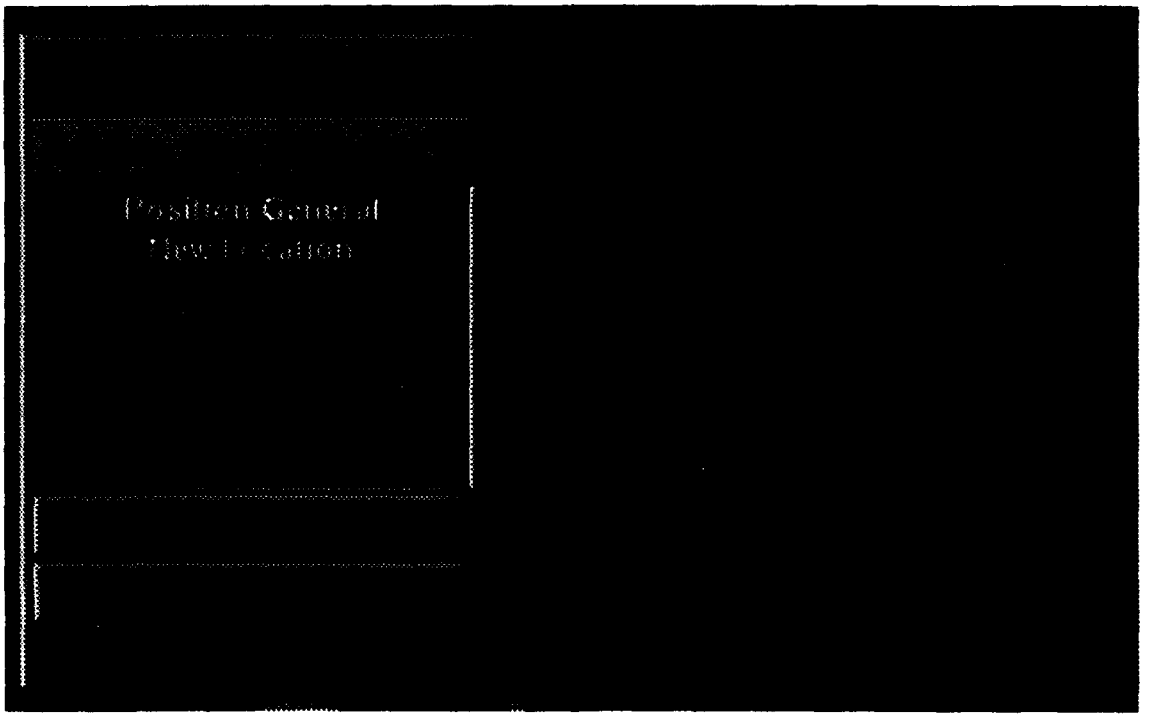


Figure I.6: Pre-Add Control Panel.

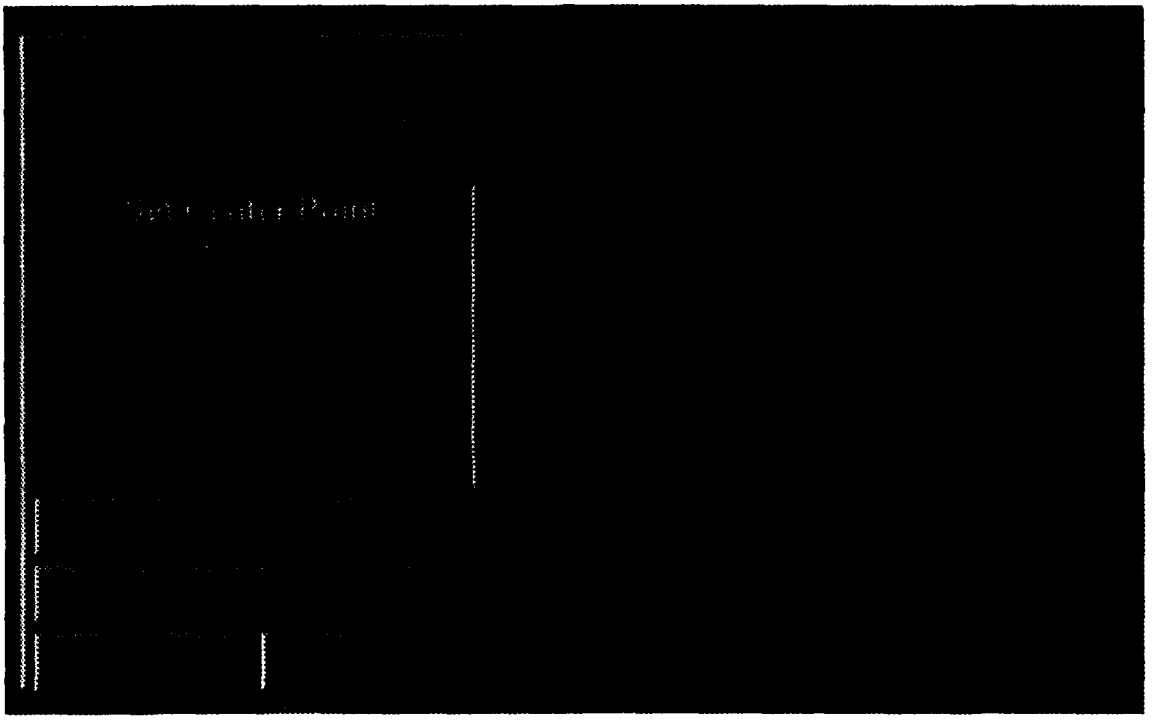


Figure I.7: During-Add Control Panel.

The virtual keyboard allows the user to enter characters into an input area with the mouse, that when ready, transfers to the input area for the Sentinel watchspace's name. The virtual keyboard can type both lower and upper case letters as well as numbers. The Clear button clears the input window and the Enter button transfers the given input. There is also a backspace button (blue solid triangle pointing left) that removes one character at a time. (See Figure I.8).

3.2.2.2.2 Attachment Control Panel

When a user is at the High Detail level control panel and attaches to a Sentinel watchspace, the High Detail level control panel adds a row of buttons that give the user more functional control over the Sentinel players. Currently there are six added functions that the user can now perform at this control level. The user can move a watchspace, modify a watchspace's radius, display a entity capability contour for the entities currently within that watchspace, display a assessment history for that watchspace, reset to the initial

view for that watchspace, and delete that watchspace (see Figure I.9). The following subsections talk about each of the above mentioned functions that contain a control panel: move and modify radius. The rest of the functions are self explanatory.

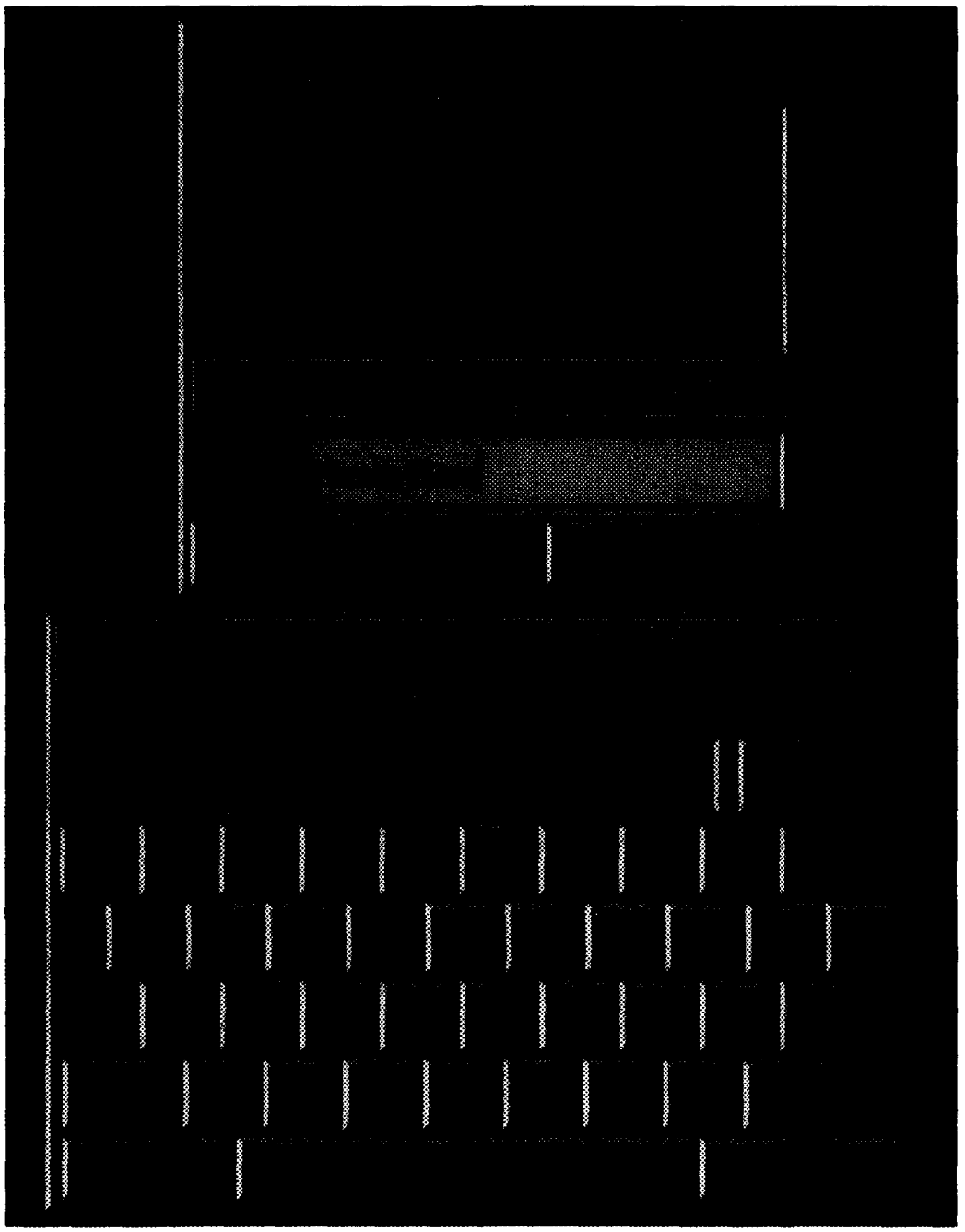


Figure I.8: Name Change and Virtual Keyboard Control Panels.

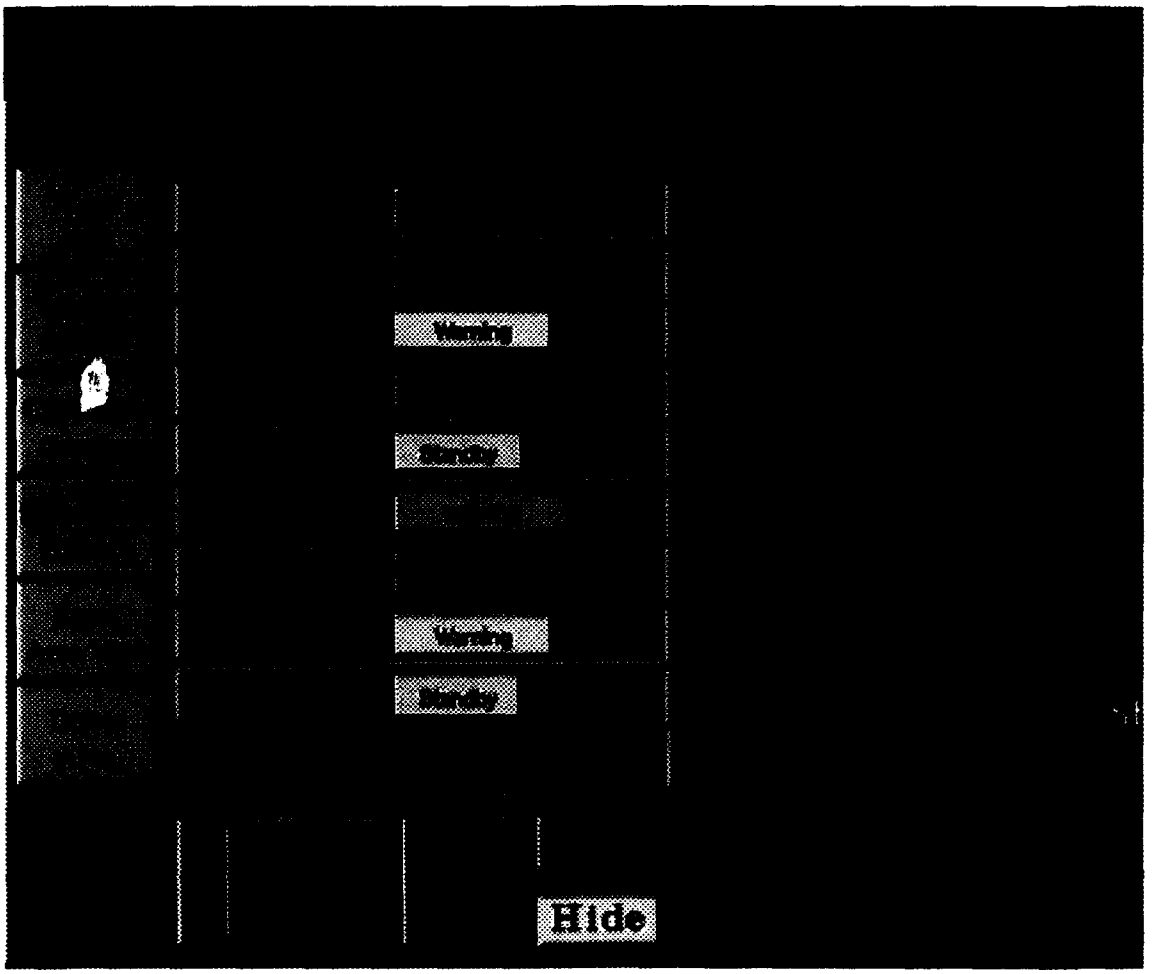


Figure I.9: Attached Control Level Control Panel.

3.2.2.2.2.1 Move Watchspace

The Move Watchspace and Add Watchspace functions (see Figures I.6 and I.7) behave in much the same way. The difference is that the Move Watchspace function works with an already defined Sentinel watchspace, while the Add Watchspace function creates a temporary Sentinel watchspace to move and place in the simulation. With this in mind, the move watchspace function also has a pre-move and a during-move control panel associated with the function. These are very similar to the ones used for the add watchspace function. The only difference is that with the move watchspace function, the user is not required to rename the moved Sentinel watchspace. Figures I.10 and I.11 show the pre-move and during-move control panels.

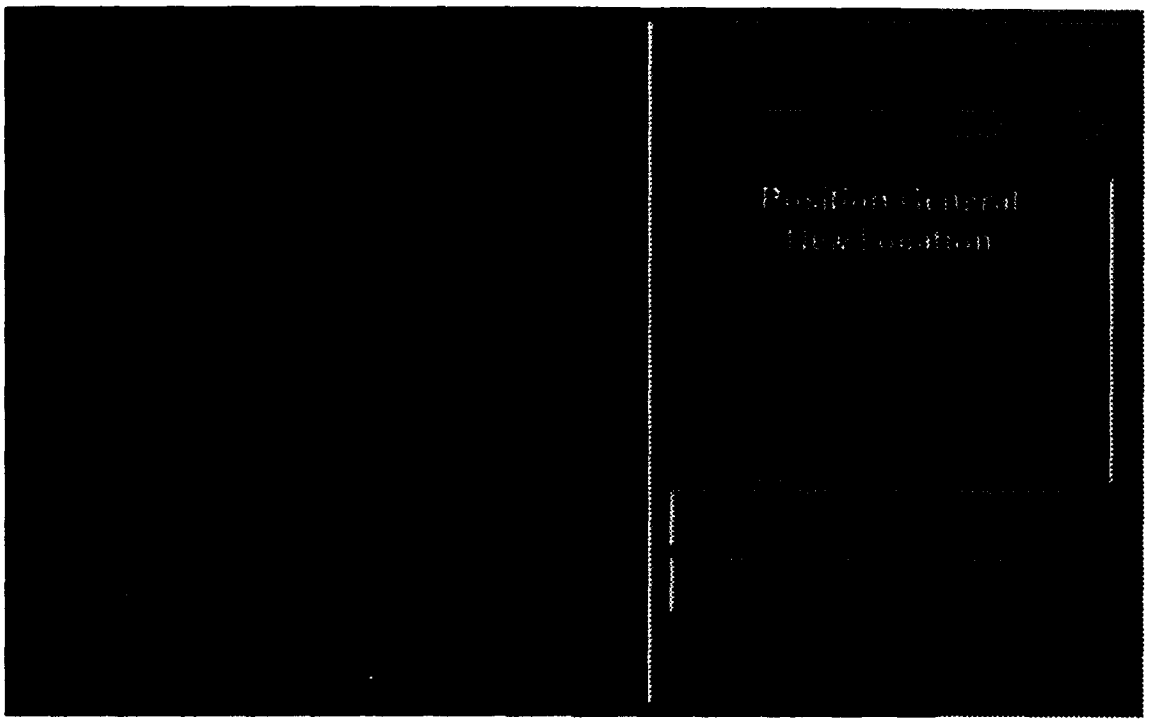


Figure I.10: Pre-Move Control Panel.

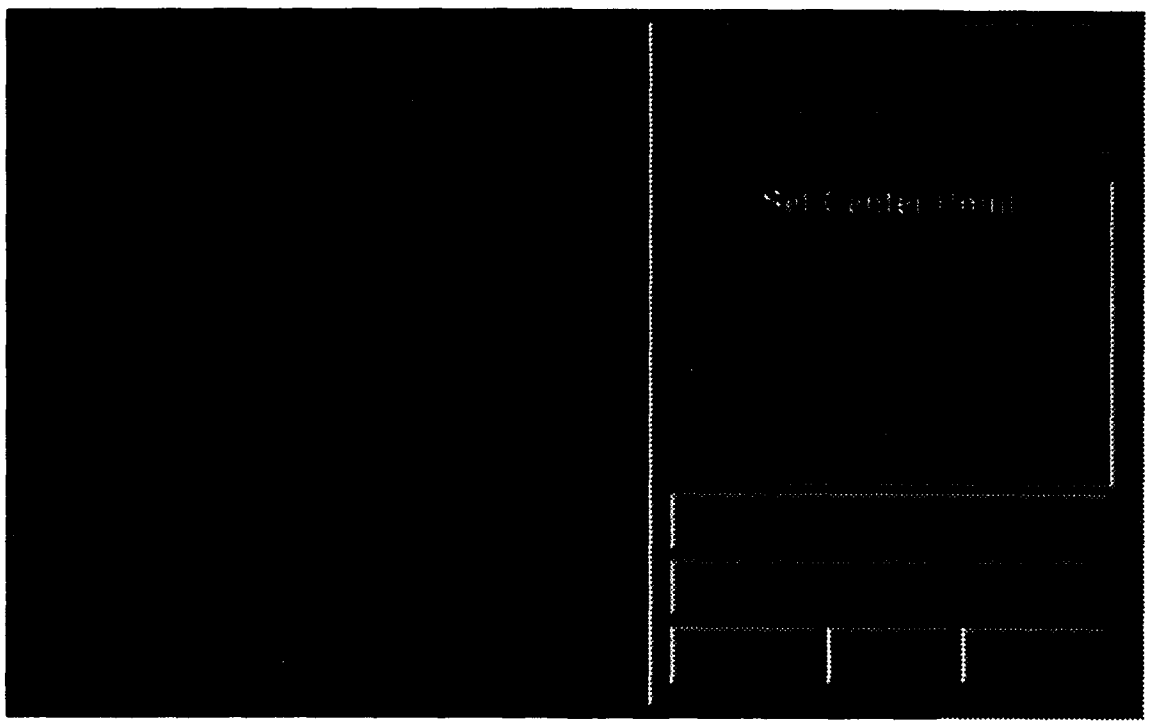


Figure I.11: During-Move Control Panel.

3.2.2.2.2.2 Modify Watchspace Radius

This function allows the user to modify the radius of the selected Sentinel watchspace through the use of the Modify Radius control panel as shown in Figure I.12.

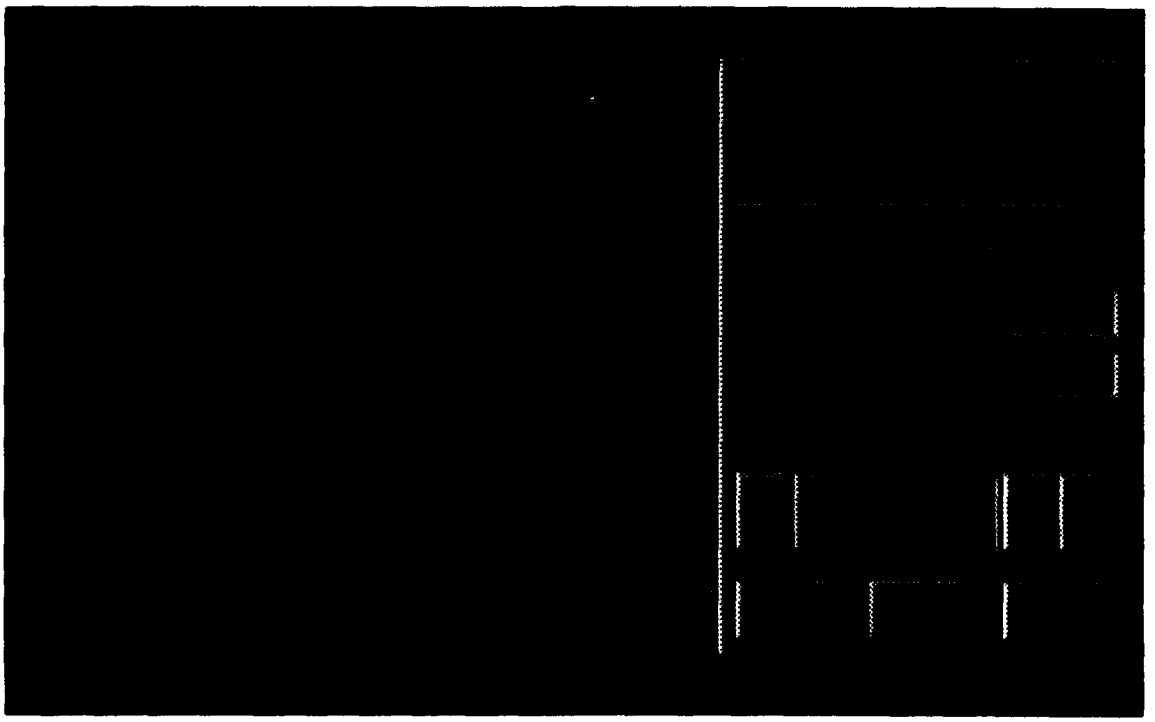


Figure I.12: Modify Radius Control Panel.

3.3 Shutdown

Since the Sentinel system is an extension of the driving application, in this case the SBB, all that has to be done to shutdown the system is to quit the driving application.

APPENDIX II: PROGRAMMERS MANUAL

1. Overview

This programmers manual tries to give the reader an understanding of how and where to make programming changes within the Sentinel system. It is by no way all inclusive, but should give enough information to get started. The manual is broken up into five sections:

- Hardware and software requirements,
- Directory structure,
- Programming particulars,
- Integration with the Synthetic BattleBridge (SBB),
- Compiling and linking.

2. Requirements

2.1 Hardware Requirements

There are currently two such workstation types for which the system can be run on:

- The Silicon Graphics IRIS 4D/440VGXT Workstation with two or more processors,
- The Silicon Graphics Onyx RealityEngine^{2TM} Workstation with two or more processors.
- Macintosh Quadra 800 or IIci for sound interface if required.

2.2 Software Requirements

2.2.1 Commercial Software Requirements

- The software was written in C++. It can be compiled with the AT&T C++, version 2.1 or 3.0.1, compilers.
- The two workstations above can be equipped with either the version 4.0.x or 5.x Silicon Graphics Operating System.
- The rendering is done with Performer 2.1.

2.2.2 Non-Commercial Software Requirements

- The driving application: SBB ([Wil93])
- The user interface software: Forms 2.1 ([Ove92])
- ObjectSim application framework ([Sny93])
- Network interface software: Object Manager ([She92])
- Model Manager ([Wil93])
- Sound Generation Facility (SGF) ([Sol92])

3. Directory Structure

Figure II.2 shows pictorially the directory structure of the Sentinel system along with the driving application (SBB). It should be noted that the root location, leo2, may have changed since the end date of this thesis work. However, the rest of the directory structure should still be intact from the bsoltz directory on.

Table II.1 gives the name of each directory shown in Figure II.1 along with a short description of its contents.

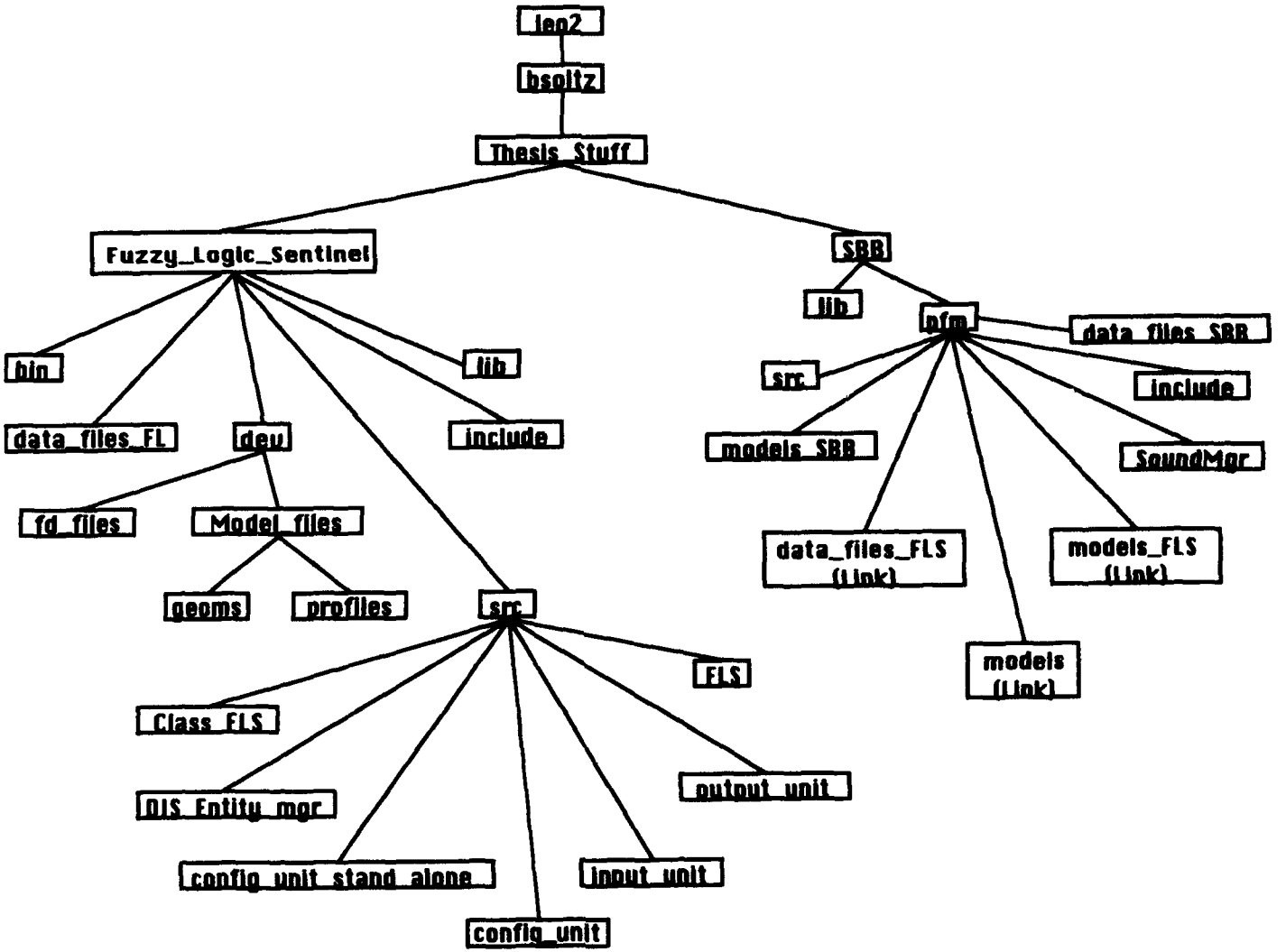


Figure II.1: Thesis Directory Structure

Table II.1 Directory Description

Directory Name	Description
leo2	place at root
bsoltz	starting place for my work
Thesis_Stuff	starting place for thesis work
Fuzzy_Logic_Sentinel	starting place for Fuzzy Logic Sentinel work
bin	holds executables for Sentinel utilities
data_files_FLS	hold data files related to the Sentinel system
dev	starting point for the development work
fd_files	basic specs for the user forms of the Sentinel
model_files	holds developed model files
geom	hold geometry for model files
profile	holds profile info for model files
include	holds the includes files for all Sentinel source code
lib	holds library file created by source code
models_FLS	holds current models used by Sentinel system
src	starting point for all source code sub-directories
DIS_Entity_mgr	source code for the DIS Entity enumerator utility
config_unit_stand_alone	source code for the watchspace and weight editor utility
Class_FLS	source code for the Sentinel Classes
config_unit	source code for the configuration unit
input_unit	source code for the input unit
FLS	source code for the fuzzy logic computation unit
output_unit	source code for the output and control units

Table II.1 Directory Description (cont.)

Directory Name	Description
SBB	starting point for the Synthetic BattleBridge (SBB)
lib	SBB dependent libraries
pfmr	starting point for the SBB code
SoundMgr	code for the Mac Sounds Class
include	holds the includes files for all SBB code
src	holds the source files for the SBB
data_files_SBB	hold data files related to the SBB system
models_SBB	holds current models used by the SBB system
data_files_FLS	link to Fuzzy_Logic_Sentinel/data_files_FLS directory
models_FLS	link to Fuzzy_Logic_Sentinel/models_FLS directory
models	link to usr/people/wb/models that has all the entity models

4. Programming Particulars

4.1 Global Data Structures

structure to hold information on each entity weight

```
typedef struct object_weights
{
   float weight;        //weight associated with this particular entity
   char name[40];       //entity name
} object_weights;
typedef object_weights All_Objects_Weights[MAX_OBJECT_TYPES];
```

structure to hold data on players watchspace location in simulation

```
struct FLS_obj_spec
{
   int friend_flag;        //friend or foe
   pfVec3 xyz;             //position
   float obj_weight;       //entity weight
   int FLS_area_number;        //watchspace number entity is in
};
```

structure to hold information on each defined area

```
struct area_specs
{
   int valid_flag;         //is the watchspace currently valid
   int lat_deg, lat_min, lat_sec;
   char lat_direction;   //N, n, S, s
   double lat_location_double;          //decimal latitude
   int lon_deg, lon_min, lon_sec;
   char lon_direction;  //E, e, W, w
   double lon_location_double;          //decimal longitude
   float radius_miles;
   float radius_meters;
   double area_size_miles;      //(PIE)(radius)(radius_miles)
   char area_name[40];          //watchspace identifier
   double x, y, z;      //position
};
```

```
const int MAX_AREAS = 10;  //Current Max number of areas

structure to hold the Current configuration data
struct input_values
{
   float  infantry_friend, infantry_en,
          armor_friend, armor_en,
          combat_ac_friend, combat_ac_en,
          combat_heli_friend, combat_heli_en,
          aaa_sam_friend, aaa_sam_en,
          artillery_friend, artillery_en,
          smart_bombs_friend, smart_bombs_en,
          jammers_friend, jammers_en;      //Maximum fuzzy category values
   char areas[5];      //number of areas as a charcacter
   char area_definitions[500];      //all watchspace definitions as a string
   int num_areas;
   area_specs area_defs[MAX_AREAS];      //array of MAX_AREAS watchspaces
   FLS_obj_spec contained_in_array[MAX_PLAYERS];//containment entities array
};

int Stand_Alone;     Flag to indicated if the Fuzzy Configuration Form is being
                     used as a stand alone application in within another application.
int Protocol_Flag;   Gives the current protocol being used.
input_values current_config;          Holds the global structure of the Sentinel
All_Objects_Weights obj_weight_array;  Holds weight information on the entities.
```

4.2 Global Defines and Constants

Table II.2 gives all the defines and constants for the Sentinel system for easy reference:

Table II.2 #defines and const

File Found In	Kind	Name	Value
FLSS_config_unit.cc	#define	AREAS	0
FLSS_SA_config_unit.cc	#define	AREAS	0
FLSS_config_unit.cc	#define	AREA_DEFINITIONS	1
FLSS_SA_config_unit.cc	#define	AREA_DEFINITIONS	1
FLSS_config_unit.cc	#define	ARMOR_E	9
FLSS_SA_config_unit.cc	#define	ARMOR_E	9
FLSS_config_unit.cc	#define	ARMOR_F	8
FLSS_SA_config_unit.cc	#define	ARMOR_F	8
FLSS_config_unit.cc	#define	ARTILLERY_E	15
FLSS_SA_config_unit.cc	#define	ARTILLERY_E	15
FLSS_config_unit.cc	#define	ARTILLERY_F	14
FLSS_SA_config_unit.cc	#define	ARTILLERY_F	14
StringClass.h	#define	BLANK	''
FLSS_config_unit.cc	#define	CAC_E	1
FLSS_SA_config_unit.cc	#define	CAC_E	1
FLSS_config_unit.cc	#define	CAC_F	0
FLSS_SA_config_unit.cc	#define	CAC_F	0
FLSS_iu_des_check.cc	#define	CASE_DEFAULT_PRINT	FALSE
FLS_membership_computations.cc	#define	CHECK_OUTPUT	FALSE
FLS_membership_computations.cc	#define	CHUNKING_COMPUTE_DEBUG	FALSE
FLS_output_color.cc	#define	COLOR_DEBUG	FALSE
FLS_chunking_definitions.h	#define	COMMENT	((char)0x23)
FLS_rule_definitions.h	#define	COMMENT	((char)0x23)
fuzzy_logic_sentinel.h	#define	COMMENT	((char)0x23)
FLS_membership_computations.cc	#define	COMPUTE_CATEGORY_MEMBERSHIP_DEBUG	FALSE
FLS_help.cc	#define	CONTROLLER	28

Table II.2 #defines and const (cont)

File Found In	Kind	Name	Value
FLS_SA_help.cc	#define	CONTROLLER	28
FL_Sentinel_mgr.cc	#define	DEBUG_COUNT	FALSE
FLSS_config_unit.cc	#define	DEFAULT_AREAS_CONFIG_FILE	"data_files_FLS/default_areas.dat"
FLSS_SA_config_unit.cc	#define	DEFAULT_AREAS_CONFIG_FILE	"data_files_FLS/default_areas.dat"
FLSS_config_unit.cc	#define	DEFAULT_AREAS_CONFIG_FILE_NAME	"default_areas.dat"
FLSS_SA_config_unit.cc	#define	DEFAULT_AREAS_CONFIG_FILE_NAME	"default_areas.dat"
FLS_chunking_definitions.h	#define	Default_Chunking_Rules_File	"data_files_FLS/chunking_rules"
fuzzy_logic_sentinel.h	#define	Default_Membership_Values_File	"data_files_FLS/ fuzzy_set_membership_default"
FLSS_output_unit.cc	#define	DEFAULT_MILES	1.0
FLSS_config_unit.cc	#define	DEFAULT_OBJ_WEIGHTS_CONFIG_FILE	"data_files_FLS/ default_obj_weights.dat"
FLSS_SA_config_unit.cc	#define	DEFAULT_OBJ_WEIGHTS_CONFIG_FILE	"data_files_FLS/ default_obj_weights.dat"
FLSS_config_unit.cc	#define	DEFAULT_OBJ_WEIGHTS_CONFIG_FILE_NAME	"default_obj_weights.dat"
FLSS_SA_config_unit.cc	#define	DEFAULT_OBJ_WEIGHTS_CONFIG_FILE_NAME	"default_obj_weights.dat"
FLS_output_rules.h	#define	Default_Output_Rules_File	"data_files_FLS/fuzzy_set_output_rules"
FLS_rule_definitions.h	#define	Default_Rules_Values_File	"data_files_FLS/fuzzy_set_rules"
FLS_SBB_interface.cc	#define	DELTA_DEBUG	FALSE
wgs84.h	const	double constE2=	2.0*flattening-flattening*flattening;
wgs84.h	const	double flattening=	1.0/298.257223563;
wgs84.h	const	double majRadius=	6378137.0;
wgs84.h	const	double minRadius=	6356752.3142;
wgs84.h	const	double radiusRatio=	(minRadius*minRadius)/(majRadius*majRadius)
fuzzy_logic_sentinel.h	#define	ECHO_PRINTER	FALSE
fuzzy_logic_sentinel.h	#define	EIGHT	((char)0x38)
FLS_chunking_definitions.h	#define	ENDOFSTRING	((char)'\0')
FLS_rule_definitions.h	#define	ENDOFSTRING	((char)'\0')
fuzzy_logic_sentinel.h	#define	ENDOFSTRING	((char)'\0')

Table II.2 #defines and const (cont)

File Found In	Kind	Name	Value
fuzzy_logic_sentinel.h	#define	EOF	(-1)
FLS_chunking_definitions.h	#define	EOL	((char)0x0A)
FLS_rule_definitions.h	#define	EOL	((char)0x0A)
fuzzy_logic_sentinel.h	#define	EOL	((char)0x0A)
FLS_membership_computations.cc	#define	EXTRA_MEMBERSHIP_COMPUTE_DEBUG	FALSE
FLSS_globals.h	#define	FALSE	0
FLS_membership_computations.cc	#define	COMPUTE_MEMBERSHIP_FUNCTION_DEBUG	FALSE
fuzzy_logic_sentinel.h	#define	FALSE	0
StringClass.h	#define	FALSE	0
wgs84.h	#define	FILTER	"/lib/cpp%slgprfilter
FLS_membership_computations.cc	#define	FINAL_OUTPUT_DEBUG	FALSE
fuzzy_logic_sentinel.h	#define	FIVE	((char)0x35)
FLSS_typdefs.h	const	float MAX_RADIUS=	50.0;
FLSS_typdefs.h	const	float TERRAIN_HEIGHT=	50000.0;
FLSS_typdefs.h	const	float TERRAIN_WIDTH=	25000.0;/
FLS_member_weight_rules.cc	#define	FLOAT_DEBUG	FALSE
FLS_SBB_interface.cc	#define	FLOAT_DEBUG	FALSE
FLSS_config_unit.cc	#define	FLSC_CANCEL_HIDE	4
FLSS_config_unit.cc	#define	FLSC_DIS_FE	0
FLSS_SA_config_unit.cc	#define	FLSC_DIS_FE	0
FL_Sentinel_mgr.cc	#define	FLSC_DIS_FE	0
FLSS_config_unit.cc	#define	FLSC_DIS_RE	1
FLSS_SA_config_unit.cc	#define	FLSC_DIS_RE	1
FL_Sentinel_mgr.cc	#define	FLSC_DIS_RE	1
FLSS_config_unit.cc	#define	FLSC_HELP	3
FLSS_SA_config_unit.cc	#define	FLSC_HELP	3
FLSS_config_unit.cc	#define	FLSC_LOAD	1
FLSS_SA_config_unit.cc	#define	FLSC_LOAD	1
FLSS_config_unit.cc	#define	FLSC_RESET	2

Table II.2 #defines and const (cont)

File Found In	Kind	Name	Value
FLSS_config_unit.cc	#define	FLSC_SAVE	0
FLSS_config_unit.cc	#define	FLSC_SAVE_HIDE	3
FLSS_SA_config_unit.cc	#define	FLSC_SAVE_TO_FILE	0
FL_Sentinel_mgr.h	#define	FLS_FORM_X	0
FL_Sentinel_mgr.h	#define	FLS_FORM_Y	0
FL_Sentinel_mgr.h	#define	FLS_ICON_X	0
FL_Sentinel_mgr.h	#define	FLS_ICON_Y	0
FL_Sentinel_mgr.h	#define	FLS_INTERVAL	5
fuzzy_logic_sentinel.cc	#define	FLS_INTERVAL	5
FLSS_output_unit.cc	#define	FLS_PLAYER_HEIGHT	10000.0
FLS_player.cc	#define	FLS_PLAYER_HEIGHT	10000.0
fuzzy_logic_sentinel.h	#define	FOUR	((char)0x34)
FLS_help.cc	#define	GENERAL	26
FLS_SA_help.cc	#define	GENERAL	26
fuzzy_logic_sentinel.h	#define	GLOBAL_DEBUG	FALSE
FLSS_output_unit.cc	#define	GO_MIN	0.80
FLSS_config_unit.cc	#define	GUIDED_MUNITIONS_E	13
FLSS_SA_config_unit.cc	#define	GUIDED_MUNITIONS_E	13
FLSS_config_unit.cc	#define	GUIDED_MUNITIONS_F	12
FLSS_SA_config_unit.cc	#define	GUIDED_MUNITIONS_F	12
FLSS_config_unit.cc	#define	HELICOPTERS_E	3
FLSS_SA_config_unit.cc	#define	HELICOPTERS_E	3
FLSS_config_unit.cc	#define	HELICOPTERS_F	2
FLSS_SA_config_unit.cc	#define	HELICOPTERS_F	2
FLSS_output_unit.cc	#define	HIGH_ALT_SCALE	10.0
wgs84.h	#define	INBUFFSIZE	256
FLSS_config_unit.cc	#define	INFANTRY_E	7
FLSS_SA_config_unit.cc	#define	INFANTRY_E	7
FLSS_config_unit.cc	#define	INFANTRY_F	6

Table II.2 #defines and const (cont)

File Found In	Kind	Name	Value
FLSS_SA_config_unit.cc	#define	INFANTRY_F	6
FLS_help.cc	#define	INPUT	27
FLS_SA_help.cc	#define	INPUT	27
FLSS_input_unit.h	const	int MAX_SUBAREAS=	10;
FLSS_output_unit.cc	const	int MAX_HISTORY	=51
FLSS_typdefs.h	const	int MAX_AREAS=	10;
FLSS_typdefs.h	const	int MAX_AREA_NAME_LENGTH=	10;
FL_Sentinel_mgr.cc	const	int SCREEN_CENTER_X	=640;
FL_Sentinel_mgr.cc	const	int SCREEN_CENTER_Y	=512;
FL_Sentinel_mgr.cc	const	int SCREEN_HEIGHT	=1024;
FL_Sentinel_mgr.cc	const	int SCREEN_WIDTH	=1280;
FLS_help.cc	#define	INTERFACE	30
FLS_SA_help.cc	#define	INTERFACE	30
FLS_SBB_interface.cc	#define	INTERFACE_DEBUG	FALSE
FLSS_output_unit.cc	#define	LABEL_SIZE	9
FLSS_config_unit.cc	//#define	LATMIN	30
FLSS_SA_config_unit.cc	//#define	LATMIN	30
FLSS_config_unit.cc	#define	LATORG	33.5 //North
FLSS_SA_config_unit.cc	#define	LATORG	33.5 //North
FLS_member_table.cc	#define	LOADER_DEBUG	FALSE
FLSS_config_unit.cc	//#define	LONGMIN	30
FLSS_SA_config_unit.cc	//#define	LONGMIN	30
FLSS_config_unit.cc	#define	LONGORG	39.5 //East
FLSS_SA_config_unit.cc	#define	LONGORG	39.5 //East
fuzzy_logic_sentinel.h	#define	Maximum_Category_Name_Length	32
FLS_chunking_definitions.h	#define	Maximum_Chunking_Rule_Name_Length	32
FLS_chunking_definitions.h	#define	Maximum_Number_Of_Chunking_Membership_Rules	5
FLS_chunking_definitions.h	#define	Maximum_Number_Of_Chunking_Rules	4
FLS_rule_definitions.h	#define	Maximum_Ruleset_Name_Length	32

Table II.2 #defines and const (cont)

File Found In	Kind	Name	Value
fuzzy_logic_sentinel.h	#define	Maximum_Table_Name_Length	32
FLS_chunking_definitions.h	#define	Maxumum_Membership_Name_Length	32
FLSS_typdefs.h	#define	MAX_PLAYERS	2000
FLS_membership_computations.cc	#define	MEMBERSHIP_COMPUTE_DEBUG	FALSE
FLSS_config_unit.cc	#define	METERS_EW	274000
FLSS_SA_config_unit.cc	#define	METERS_EW	274000
FLSS_config_unit.cc	#define	METERS_NS	115000
FLSS_SA_config_unit.cc	#define	METERS_NS	115000
FLSS_config_unit.cc	#define	METERS_PER_MIN	1853
FLSS_SA_config_unit.cc	#define	METERS_PER_MIN	1853
FLSS_output_unit.cc	#define	MID_ALT_SCALE	5.0
FLSS_config_unit.cc	#define	MISC_AIR_E	5
FLSS_SA_config_unit.cc	#define	MISC_AIR_E	5
FLSS_config_unit.cc	#define	MISC_AIR_F	4
FLSS_SA_config_unit.cc	#define	MISC_AIR_F	4
FLSS_config_unit.cc	#define	MISC_GROUND_E	11
FLSS_SA_config_unit.cc	#define	MISC_GROUND_E	11
FLSS_config_unit.cc	#define	MISC_GROUND_F	10
FLSS_SA_config_unit.cc	#define	MISC_GROUND_F	10
FLSS_config_unit.cc	#define	MISC_MUNITION_E	17
FLSS_SA_config_unit.cc	#define	MISC_MUNITION_E	17
FLSS_config_unit.cc	#define	MISC_MUNITION_F	16
FLSS_SA_config_unit.cc	#define	MISC_MUNITION_F	16
FLS_chunking_definitions.h	#define	NEWLINE	((char)'\n')
FLS_rule_definitions.h	#define	NEWLINE	((char)'\n')
fuzzy_logic_sentinel.h	#define	NEWLINE	((char)'\n')
StringClass.h	#define	NEWLINE	'\n'
fuzzy_logic_sentinel.h	#define	NINE	((char)0x39)
FLS_chunking_definitions.h	#define	NUL	((char)0)

Table II.2 #defines and const (cont)

File Found In	Kind	Name	Value
FLS_rule_definitions.h	#define	NUL	((char)0)
fuzzy_logic_sentinel.h	#define	NUL	((char)0)
FLSS_globals.h	#define	NULL	0
fuzzy_logic_sentinel.h	#define	Number_of_Categories	8
FLS_output_rules.h	#define	Number_Of_Chunked_Values	10
FLS_interface_structures.h	#define	Number_of_Color_Cue_Levels	11
fuzzy_logic_sentinel.h	#define	Number_of_Membership_Tables	4
FLS_output_rules.h	#define	Number_Of_Output_Rules	36
FLS_rule_definitions.h	#define	Number_Of_Rulesets	1
FLS_rule_definitions.h	#define	Number_of_Rules	Number_of_Membership_Tables
FLS_defines.h	#define	Number_Of_Subspaces	10
fuzzy_logic_sentinel.h	#define	Number_Of_Subspaces	10
fuzzy_logic_sentinel.h	#define	ONE	((char)0x31)
FLS_help.cc	#define	OUTPUT	29
FLS_SA_help.cc	#define	OUTPUT	29
FLS_interface_structures.h	#define	Output_Color_Defs	"data_files_FLS/output_color_cues_file"
FLS_output_rules.cc	#define	OUTPUT_RULES_DEBUG	FALSE
FLS_chunking_definitions.h	#define	PERIOD	((char)0x2E)
FLS_rule_definitions.h	#define	PERIOD	((char)0x2E)
fuzzy_logic_sentinel.h	#define	PERIOD	((char)0x2E)
FLSS_config_unit.cc	#define	PIE	3.1415927
FLSS_output_unit.cc	#define	PIE	3.1415927
FLSS_SA_config_unit.cc	#define	PIE	3.1415927
FLS_membership_computations.cc	#define	PRESENCE_WEIGHT	((double)2.00)
FLSS_config_unit.cc	#define	PUSHED	1
FLSS_SA_config_unit.cc	#define	PUSHED	1
FLS_help.cc	#define	PUSHED	1
FLS_help.cc	#define	QUIT_HELP	24
FLS_SA_help.cc	#define	QUIT_HELP	24

Table II.2 #defines and const (cont)

File Found In	Kind	Name	Value
FLS_chunking_definitions.h	#define	QUOTE	((char)0x22)
FLS_rule_definitions.h	#define	QUOTE	((char)0x22)
fuzzy_logic_sentinel.h	#define	QUOTE	((char)0x22)
FLSS_output_unit.cc	#define	RANDOM_ON	FALSE
PipeClass.h	#define	READMODE	"r"
FLSS_config_unit.cc	#define	RELEASED	0
FLSS_SA_config_unit.cc	#define	RELEASED	0
FLS_help.cc	#define	RELEASED	0
FLS_chunking_rules.cc	#define	RULES_DEBUG	FALSE
FLS_member_weight_rules.cc	#define	RULES_DEBUG	FALSE
fuzzy_logic_sentinel.h	#define	SEVEN	((char)0x37)
fuzzy_logic_sentinel.h	#define	SIX	((char)0x36)
FLS_rule_definitions.h	#define	Size_Of_Rules_Table	(Number_of_Categories*2)
FLS_chunking_definitions.h	#define	SPACE	((char)0x20)
FLS_rule_definitions.h	#define	SPACE	((char)0x20)
fuzzy_logic_sentinel.h	#define	SPACE	((char)0x20)
FLSS_output_unit.cc	#define	STANDARD_ALT_SCALE	1.25
FLSS_output_unit.cc	#define	STANDBY_MIN	0.45
FLS_chunking_definitions.h	#define	TAB	((char)'\t')
FLS_rule_definitions.h	#define	TAB	((char)'\t')
fuzzy_logic_sentinel.h	#define	TAB	((char)'\t')
FLS_help.cc	#define	TESTING	31
FLS_SA_help.cc	#define	TESTING	31
fuzzy_logic_sentinel.h	#define	THREE	((char)0x33)
FLSS_globals.h	#define	TRUE	1
FLSS_iu_initializer.cc	#define	TRUE	1
fuzzy_logic_sentinel.h	#define	TRUE	1
StringClass.h	#define	TRUE	1
fuzzy_logic_sentinel.h	#define	TWO	((char)0x32)

Table II.2 #defines and const (cont)

File Found In	Kind	Name	Value
fuzzy_logic_sentinel.h	#define	UNDERLINE	((char)0x5f)
FLS_chunking_rules.cc	#define	VERIFY_CHUNKING_RULES	FALSE
FLS_member_weight_rules.cc	#define	VERIFY_RULES_CONTENTS	FALSE
FLS_member_table.cc	#define	VERIFY_TABLE_CONTENTS	FALSE
FLS_help.cc	#define	VERSION	25
FLS_SA_help.cc	#define	VERSION	25
FLSS_output_unit.cc	#define	WARNING_MIN	0.60
FLSS_config_unit.cc	#define	WEIGHT_BRO_RETURN	2
FLSS_SA_config_unit.cc	#define	WEIGHT_BRO_RETURN	2
FLSS_config_unit.cc	#define	WEIGHT_BRO_SAVE	0
FLSS_SA_config_unit.cc	#define	WEIGHT_BRO_SAVE	0
FLSS_config_unit.cc	#define	WEIGHT_BRO_SAVE_RETURN	1
FLSS_SA_config_unit.cc	#define	WEIGHT_BRO_SAVE_RETURN	1
FLSS_config_unit.cc	#define	WEIGHT_SELECTOR	3
FLSS_SA_config_unit.cc	#define	WEIGHT_SELECTOR	3
wgs84.h	#define	WGS84DATA	"data_files_FLS/wgs84.dat"
PipeClass.h	#define	WRITEMODE	"w"
FLSS_config_unit.cc	#define	XYZ_FLAT	TRUE
FLSS_config_unit.cc	#define	XYZ_FLAT_CONFIG_FILE	"data_files_FLS/xyz_FE_default_areas.dat"
fuzzy_logic_sentinel.h	#define	ZERO	((char)0x30)

4.3 Structured Programming Unit Procedures

The following subsections have a table for each structured program unit that gives the names of all the procedures located in that unit. The tables give the name of the file the procedure can be found in and the name of the procedure and its input and output parameters. Functional descriptions of the procedures can be found in the code itself.

4.3.1 Configuration Unit

Table II.3 Configuration Unit Procedures

File Name	Procedure Name
FLSS_config_unit.cc	void reset_weight_btns()
FLSS_config_unit.cc	void reset_control_btns()
FLSS_config_unit.cc	void set_protocol_btn()
FLSS_config_unit.cc	void load_area_obj_weight_default_files()
FLSS_config_unit.cc	void load_obj_weight_file()
FLSS_config_unit.cc	void set_all_obj_weights_to_one()
FLSS_config_unit.cc	void save_to_weight_file()
FLSS_config_unit.cc	int lat_lon_error_check(int i, area_specs *area_defs, int default_files)
FLSS_config_unit.cc	void convert_xyz_FLAT_EARTH(double lat_location_double, double lon_location_double, double *x, double *y, double *z)
FLSS_config_unit.cc	void setup_XYZ_FLAT_areas()
FLSS_config_unit.cc	int save_default_values(input_values default_values, All_Objects_Weights temp_obj_weight_array)
FLSS_config_unit.cc	void save_obj_weights()
FLSS_config_unit.cc	void FLSC_Panel_Control_cb(FL_OBJECT *obj, long item)
FLSS_config_unit.cc	void load_weight_bro(char *group_title, int start_index, int stop_index)
FLSS_config_unit.cc	void load_weight_browser_cb(FL_OBJECT *obj, long item)
FLSS_config_unit.cc	void weight_bro_control_cb(FL_OBJECT *obj, long item)
FLSS_config_unit.cc	void FLSC_Protocol_cb(FL_OBJECT *obj, long item)
FLSS_config_unit.cc	void Toggle_Config_cb(FL_OBJECT *obj, long item)
FLSS_config_unit.cc	void Initialize_FLSC_Forms()
FLSS_config_unit.cc	void Config_Form_Constructor()
FLSS_config_forms.cc	void create_form_Fuzzy_Configure_Info(void)
FLSS_config_forms.cc	void create_form_error_browser(void)
FLSS_config_forms.cc	void create_form_helpform(void)
FLSS_config_forms.cc	void create_form_weight_browser(void)
FLSS_config_forms.cc	void create_form_Config_Form_Hidden(void)
FLSS_config_forms.cc	void create_the_config_forms(void)

Table II.3 Configuration Unit Procedures (cont)

File Name	Procedure Name
FLS_help.cc	static void load_it(char str[][90])
FLS_help.cc	void help_cb(FL_OBJECT *obj, long arg)
FLS_help.cc	void exithelp_cb(FL_OBJECT *obj, long arg)
FLS_help.cc	void showhelp_cb(FL_OBJECT *obj, long arg)
wgs84.cc	void plh2wgs84(double phi, double lambda, double elevation, double *x, double *y, double *z)

4.3.2 Input Unit

Table II.4 Input Unit Procedures

File Name	Procedure Name
FLSS_input_unit.cc	int contained_in_FE(pfVec3 location, int subspace_num)
FLSS_input_unit.cc	int contained_in_RE(pfVec3 location, int subspace_num)
FLSS_input_unit.cc	void count_object(SBB_Net_Player* current_object, int Obj_name, All_Objects_FLS count_array, int subspace_id)
FLSS_input_unit.cc	void process_data(Incoming_FLS_Data Temp_Data, All_Objects_FLS count_array)
FLSS_iu_des_check.cc	int designation_check(int designation)
FLSS_iu_des_check.cc	int platform_munition_check(SBB_Net_Player* current_object)
FLSS_iu_des_check.cc	int lifeform_check(SBB_Net_Player* current_object)
FLSS_iu_des_check.cc	int need_object(SBB_Net_Player* current_object)
FLSS_iu_initializer.cc	void initialize_Incoming_FLS_Data(Incoming_FLS_Data Temp_Data)
FLSS_iu_initializer.cc	void initialize_count_array(All_Objects_FLS count_array)
FLSS_iu_initializer.cc	void initialize_contained_in_array()

4.3.3 Computation Unit

Table II.5 Computation Unit Procedures

File Name	Procedure Name
FLS_SBB_interface.cc	int retrieve_int_value(FILE *file_ptr)
FLS_SBB_interface.cc	float retrieve_float_value(FILE *file_ptr)
FLS_SBB_interface.cc	void get_subspace_data(Incoming_FLS_Data Temp_Data)
FLS_SBB_interface.cc	void compute_subspace_deltas()
FLS_SBB_interface.cc	void update_old_subspace_array()
FLS_chunking_rules.cc	void find_chunker_name(int current_rule, FILE *file_ptr)
FLS_chunking_rules.cc	void get_chunker_category_names(int current_rule, int number_of_categories, FILE *file_ptr)
FLS_chunking_rules.cc	void load_one_chunking_rule(int current_rule, FILE *file_ptr)
FLS_chunking_rules.cc	void print_chunk_rules(int current_rule)
FLS_chunking_rules.cc	void load_chunking_rules()
FLS_member_table.cc	void get_number_of_categories(FILE *file_ptr)
FLS_member_table.cc	int find_category_name(FILE *file_ptr, Cat_Name cat_name)
FLS_member_table.cc	int find_category_max_number(FILE *file_ptr)
FLS_member_table.cc	int find_number_of_table_entries(FILE *file_ptr)
FLS_member_table.cc	Membership_Table_Ptr get_table_memory(int number_of_table_entries)
FLS_member_table.cc	void table_loader(int cat_num, int number_of_table_entries, FILE *file_ptr)
FLS_member_table.cc	void enter_membership_table_names(int cat_num)
FLS_member_table.cc	void print_loaded_tables(int cat_num, FILE *file_ptr)
FLS_member_table.cc	void load_membership_tables()
FLS_member_weight_rules.cc	void find_ruleset_name(int ruleset_num, FILE *file_ptr)
FLS_member_weight_rules.cc	void load_membership_table_weights(int rule_num, int cat_num, FILE *file_ptr)
FLS_member_weight_rules.cc	void print_loaded_rules(int ruleset_num)
FLS_member_weight_rules.cc	void load_rules()
FLS_membership_computations.cc	float compute_membership(float current_input, Membership_Table_Entry_Ptr next_table, int array_size)

Table II.5 Computation Unit Procedures (cont)

File Name	Procedure Name
FLS_membership_computations.cc	float compute_category_membership(float current_input, int current_category, int rule_num,
FLS_membership_computations.cc	void compute_weighted_membership_values(int rule_num)
FLS_membership_computations.cc	void chunk_weighted_membership_values()
FLS_membership_computations.cc	void compute_final_output()
FLS_output_color.cc	void get_output_colors()
FLS_output_rules.cc	void output_rules_error(int i, int j, int next_int)
FLS_output_rules.cc	void load_output_rules()
FLS_readfiles.cc	int find_data(FILE *file_ptr)

4.3.4 Output and Control Unit

Table II.6 Output and Control Unit Procedures

File Name	Procedure Name
FLSS_output_forms.cc	void create_form_Threat_Form(void)
FLSS_output_forms.cc	void create_form_Threat_Form_Hidden(void)
FLSS_output_forms.cc	void create_form_Threat_Form_Low_Info(void)
FLSS_output_forms.cc	void create_form_Threat_History_Graph_Form(void)
FLSS_output_forms.cc	void create_form_Threat_Radius_Form(void)
FLSS_output_forms.cc	void create_form_Threat_Move_Form(void)
FLSS_output_forms.cc	void create_form_Threat_Add_Form(void)
FLSS_output_forms.cc	void create_form_Threat_Name_Change_Form(void)
FLSS_output_forms.cc	void create_form_Virtual_Keyboard_Form(void)
FLSS_output_forms.cc	void create_output_forms(void)
FLSS_output_unit.cc	int get_next_slot_available()
FLSS_output_unit.cc	void initialize_threat_histories()
FLSS_output_unit.cc	void clear_area_history_array(long item)
FLSS_output_unit.cc	int insert_threat_history(struct item the_item, int area_number)
FLSS_output_unit.cc	void initialize_radius_form()

Table II.6 Output and Control Unit Procedures (cont)

File Name	Procedure Name
FLSS_output_unit.cc	void reset_attach_buttons() //set all buttons released
FLSS_output_unit.cc	void attach_to_area_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void display_threat_history()
FLSS_output_unit.cc	void show_not_used_slot(long item)
FLSS_output_unit.cc	void show_slot(long item, char *temp_string)
FLSS_output_unit.cc	static void load_instructions(char str[8][40], FL_OBJECT *obj)
FLSS_output_unit.cc	void initialize_vk()
FLSS_output_unit.cc	void initialize_name_change_form(int area_index, int keyboard_flag)
FLSS_output_unit.cc	void load_upper_case()
FLSS_output_unit.cc	void load_lower_case()
FLSS_output_unit.cc	void vk_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void attach_control_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void radius_control_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void reset_default_cursor()
FLSS_output_unit.cc	void move_area_control_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void detach_from_area_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void add_area_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void name_change_area_control_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void add_area_control_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void return_btn_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void make_icon_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void more_info_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void prev_info_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void toggle_threat_cb(FL_OBJECT *obj, long item)
FLSS_output_unit.cc	void initialize_form(int num)
FLSS_output_unit.cc	char *calculate_interrupt_level(float risk)
FLSS_output_unit.cc	void calculate_color(float risk, int *red, int *green, int *blue)
FLSS_output_unit.cc	int check_area_attach_btn(int i)
FLSS_output_unit.cc	void update_FLS_area_values(Outgoing_FLS_Data outgoing_data)
FLSS_output_unit.cc	void Output_Form_Constructor()

4.4 *Object-Oriented Class Methods*

The following subsections have a table for each Object-Oriented Class that gives the names of all the methods located in that unit. The tables give the name of the file the method can be found in and the name of the method and its input and output parameters. Functional descriptions of the methods can be found in the code itself.

4.4.1 *FLS_Player Class*

Table II.7 FLS_Player Class Methods

File Name	Method Name
FLS_player.cc	FLS_Player :: FLS_Player( )
FLS_player.cc	FLS_Player :: FLS_Player (float sc_x, float sc_y, float sc_z)
FLS_player.cc	void FLS_Player :: init_index (int id)
FLS_player.cc	void FLS_Player :: set_index(int id)
FLS_player.cc	int FLS_Player :: get_view_index(int val)
FLS_player.cc	void FLS_Player :: set_position (float new_x, float new_y, float new_z)
FLS_player.cc	void FLS_Player :: set_scale (float sc_x, float sc_y, float sc_z)
FLS_player.cc	void FLS_Player :: set_delta (float d_x, float d_y, float d_z)
FLS_player.cc	void FLS_Player :: init_shared( )
FLS_player.cc	void FLS_Player :: init( )
FLS_player.cc	void FLS_Player :: init(pfGroup* FLS_cyl_group)
FLS_player.cc	void FLS_Player :: propagate(int on_off)
FLS_player.cc	void FLS_Player :: sort_ids(int *FLS_cyl_ids, FLS_Player *FLS_Cyl, Stealth *Ov, int attached_index)
FLS_player.cc	void FLS_Player :: init_sim_FLS_players (Color_View MainViewobj, FL_Sentinel FuzzySen, FLS_Player *FLS_Cyl , Stealth *Ov, int max_stealth_views, Simple_Terrain* Ter)
FLS_player.cc	void FLS_Player :: propagate_FLS_players(FL_Sentinel FuzzySen, FLS_Player *FLS_Cyl, Stealth *Ov, Simple_Terrain* Ter, int attached_index, int switch_locators, int on_off)
FLS_player.cc	void FLS_Player :: assign_space_and_initialize_FLS_players (FL_Sentinel FuzzySen, FLS_Player *FLS_Cyl, Pfmr_Renderer* Rend, Sim_Entity_Mgr* Sim, Model_Manager* Mod)

4.4.2 FL_Sentinel Class

Table II.8 FL_Sentinel Class Methods

File Name	Method Name
FL_Sentinel_mgr.cc	void FL_Sentinel :: draw ( )
FL_Sentinel_mgr.cc	void FL_Sentinel::capability_on(int area_num , int val)
FL_Sentinel_mgr.cc	input_values FL_Sentinel::get_area_defs()
FL_Sentinel_mgr.cc	void FL_Sentinel::update_area_defs(input_values current_areas)
FL_Sentinel_mgr.cc	int FL_Sentinel::count_objects(Incoming_FLS_Data Temp_Data)
FL_Sentinel_mgr.cc	void FL_Sentinel::config() //must be done in the application thread
FL_Sentinel_mgr.cc	void FL_Sentinel::area_deleted(int area_num)
FL_Sentinel_mgr.cc	void FL_Sentinel::area_radius_changed(int area_num)
FL_Sentinel_mgr.cc	void FL_Sentinel :: change_view(int area_num, float scale_altitude)
FL_Sentinel_mgr.cc	void FL_Sentinel::move_area_ready(int area_num)
FL_Sentinel_mgr.cc	void FL_Sentinel :: set_moving_attached(int moving_attach)
FL_Sentinel_mgr.cc	void FL_Sentinel::add_area_ready(int area_num)
FL_Sentinel_mgr.cc	void FL_Sentinel::add_area_here(int area_num)
FL_Sentinel_mgr.cc	void FL_Sentinel::set_highlight(int area_num, int val)
FL_Sentinel_mgr.cc	void FL_Sentinel::init(SBB_Net_Manager* Net, long iconx, long icony, long formx, long formy)
FL_Sentinel_mgr.cc	void FL_Sentinel::start_forms(long *FLSwindow, long *FLSCwindow)
FL_Sentinel_mgr.cc	void FL_Sentinel::set_icon_position(long iconx, long icony)
FL_Sentinel_mgr.cc	void FL_Sentinel::set_form_position(long formx, long formy)
FL_Sentinel_mgr.cc	void FL_Sentinel :: set_attached(int prior_attach, int current_attach)
FL_Sentinel_mgr.cc	void FL_Sentinel::update()

5. Integration With the Synthetic BattleBridge

To integrate the Sentinel system into the SBB, two source code files of the SBB have to be modified. The following code shows the places in the files that have to be changed. These files are located in ~bsoltz/Thesis_Stuff/SBB/pfmr/src.

stealth.cc

```
                                                    •
                                                    •
                                                    •
#include "drawstring.h"      // for string output
#include "button.h"          // for button output
#include "stealth.h"

////////////////////////////////////////////////
#include "FL_Sentinel_mgr.h"
extern int include_FLS;
extern FL_Sentinel FuzzySen;
////////////////////////////////////////////////

extern "C"
{
        #include "forms.h"
        #include "sbb_forms.h"
}
                                                    •
                                                    •
                                                    •
void Stealth :: draw()
{
        set_screen( 0.0, SCREEN_HEIGHT, SCREEN_WIDTH, 0.0,
                SCREEN_CENTER_X, SCREEN_CENTER_Y, SCALE_FACTOR );
        Sh->altitude        = Coords->xyz[PF_Z];
        Sh->heading         = Coords->hpr[PF_H];
        if ( Sh->current_net_list_index != -1 )
        {
                Sh->altitude -= Sh->base_offst[PF_Y];
                Sh->heading += Sh->base_rot[PF_H];
        }
```

```
////////////////////////////////////////////////
if (include_FLS)
   FuzzySen.draw();

set_screen( 0.0, SCREEN_HEIGHT, SCREEN_WIDTH, 0.0,
                 SCREEN_CENTER_X, SCREEN_CENTER_Y,
                 SCALE_FACTOR );
////////////////////////////////////////////////////

set_xform_matrix( SCREEN_CENTER_X, SCREEN_CENTER_Y, Sh->heading );
```

•
•
•

sbb_app.cc

•
•
•

```
#include "drawstring.h"
#include "button.h"
#include "sounds.h"
////////////////////////////////////////////////////
#include "FLS_player.h"
////////////////////////////////////////////////////
extern "C"
{
        #include "forms.h"
        #include "sbb_forms.h"
}
```

•
•
•

```
typedef struct
{
```

```
                                        .
                                        .
                                        .
        int mouse_x;
        int mouse_y;
        int mouse_pressed;
        ////////////////////////////////////////////////
        long FLS_forms_window_id;       //for both SBB & FLS
        long FLSC_forms_window_id;      // & FLSC ie configure forms
        int fuzz_cyl_display;           //Toggle for FLS solid cylinders
        int alt_key_pushed;             //Toggle for alt key
        int switch_fuzz_cyl_display;    //Toggle for FLS cage cylinders
        ////////////////////////////////////////////////
} Shared;
static Shared* Sh;
                                        .
                                        .
                                        .
// ////////////////////////////////////////////////////////////////////////
// forms globals
int include_Nav   = FALSE;
////////////////////////////////////////////////
////      FLS 21 Nov 93
int include_FLS       = FALSE;
////////////////////////////////////////////////
// ////////////////////////////////////////////////////////////////////////
// forms globals
int NTSC_Mode   = FALSE;
                                        .
                                        .
                                        .
// ////////////////////////////////////////////////////////////////////////
// The sounds player
static Mac_Sounds* SBB_Draw_Sound;
```

```
//////////////////////////////////////////////////////
FLS_Player FLS_Cyl[MAX_AREAS];
FL_Sentinel FuzzySen;
extern void reset_default_cursor();     //from output unit
//////////////////////////////////////////////////////
//sbb_button call_back function prototype (to sbb_button)
extern void LocButtonPressed( int val );
extern void LocButtonPressedNet( int val );
                                        .
                                        .
                                        .
void SBB_App::initialize()
{
                                        .
                                        .
                                        .
        Sh->sounds_enable               = TRUE;
        Sh->sound_chan                  = 1;
        //////////////////////////////////////////////
        Sh->fuzz_cyl_display                = TRUE;
        Sh->switch_fuzz_cyl_display         = FALSE;
        //////////////////////////////////////////////
}
                                        .
                                        .
                                        .
// This function is called from the renderer obj after Performer
// is ready for models to be added
void SBB_App::init_sim()
{
                                        .
                                        .
                                        .
        MainViewobj.new_view(0);
```

```
        ////////////////////////////////////////////////
        if (include_FLS)
          FLS_Cyl[0].init_sim_FLS_players( MainViewobj, FuzzySen,
        FLS_Cyl , Ov, MAX_STEALTH_VIEWS, &terrain);
        ////////////////////////////////////////////////
        // Now, initialize the array of stealth players/views
        for ( i = 0; i < MAX_STEALTH_VIEWS; i++ )
                Ov[i].init();
        if ( three_d_render )
                Off.init();
        if ( MainViewobj.Delta != NULL )
                MainViewobj.Delta->init();
        SBB_App_Sound = new Mac_Sounds();
}
                                                .
                                                .
                                                .
void SBB_App :: init_draw_thread()
{
                                                .
                                                .
                                                .
        fl_init();
        fl_unqdevice( INPUTCHANGE );
        if( include_Nav )
        {
                create_the_sbb_forms();
                initialize_the_sbb_forms();
                fl_set_form_position( NavFormHidden, -1, 0 );
                Sh->forms_window_id = fl_show_form( NavFormHidden,
                                FL_PLACE_POSITION, FALSE, NULL );
        }
```

```
        ////////////////////////////////////////
        if (include_FLS)
        {
          FuzzySen.init( &NetMan );          //DIS
          FuzzySen.start_forms(&Sh->FLS_forms_window_id,
                                    &Sh->FLSC_forms_window_id);
        }
        ////////////////////////////////////////////
        fl_qdevice(PADVIRGULEKEY);      // cycle backward through stealth views
        fl_qdevice(PADASTERKEY);        // cycle forward through stealth views
        fl_qdevice(NUMLOCKKEY);         // enable default stealth view
                                            .
                                            .
                                            .
}
void SBB_App :: pre_draw()
{
                                            .
                                            .
                                            .
        FL_OBJECT *obj;
        ////////////////////////////////////////
        if (include_FLS)
        {
          FuzzySen.update();
          if (FuzzySen.Sh->reset_default_cursor_flag == TRUE)
              reset_default_cursor();           //back to the arrow
        }
        ////////////////////////////////////////////////
        obj = fl_check_forms();
        while ( obj != NULL && mine )
        {
        if ( obj == FL_EVENT )
        {
              short value;
```

```
            long but = fl_qread( &value );
            if (value) switch ( but )
            {
                                        •
                                        •
                                        •
////////////////////////////////////////
//// Toggle for FLS cylinders  FLS 21 Nov 93
case F11KEY:          // Set index to default view.
      if ((getbutton(LEFTALTKEY) == TRUE) ||
        (getbutton(LEFTALTKEY) == TRUE))
      {
            Sh->switch_fuzz_cyl_display = TRUE;
      }//end if alt key pushed
      else if ((getbutton(LEFTCTRLKEY) == TRUE) ||
             (getbutton(LEFTCTRLKEY) == TRUE))
      {
            if (include_FLS)
            {// pull the FLSC forms window to the top
                  winset( Sh->FLSC_forms_window_id );
                  winpop();
                  // pull the FLS forms window to the top
                  winset( Sh->FLS_forms_window_id );
                  winpop();
            }
            // pull the forms window to the top
            if (include_Nav)
            {
                  winset( Sh->forms_window_id );
                  winpop();
            }
            winset( Sh->sbb_window_id );
      }//end if control key pushed
      else
      {
```

```
            Sh->fuzz_cyl_display = !Sh->fuzz_cyl_display;
        }//end just F11 key pushed
        break;
        ////////////////////////////////////////
                                        .
                                        .
                                        .
void SBB_App :: propagate (int& exitflag)
{
                                        .
                                        .
                                        .
            ////////////////////////////////////
            if (include_FLS)
            {
              FLS_Cyl[0].propagate_FLS_players( FuzzySen, FLS_Cyl,
            Ov, &terrain, attached_index,
                    Sh->switch_fuzz_cyl_display, Sh->fuzz_cyl_display);
            }
            Sh->switch_fuzz_cyl_display = FALSE;
            ////////////////////////////////////////
            SimMan.update_state();
            if( Sh->clip_flag )
            {
                    ClipPlaneSet( Sh->clip_index );
            }
        }
        else
        {
            NetMan.kill_net();
        }
}
                                        .
                                        .
                                        .
```

```
int parse_command ( int argc, char *argv[] )
{
        // command line option
        int  option;
        int returnval = TRUE;
        cerr << "SBB setup:" << "\n";
        while ( ( option =  getopt( argc, argv, "d:3gblsfv:tNnzh" ) ) != -1)
        {
                                                .
                                                .
                                                .
                        case 'z':       // FLS form  (for FLS app)
                                include_FLS = TRUE;
                                cerr << "\t*** " << "enabled: FLS" << "\n";
                                break;
                        case 'h':  // help
                                cerr << USAGE << "\n";
                                returnval = FALSE;
                                break;
                        default:
                                break;
                }
        }
        return ( returnval );
}
                                                .
                                                .
                                                .
int main (int argc, char *argv[])
{
        int exitflag = 0;
        int  next;
        ////////////////////////////////////////////////
        Mac_Sounds* SBB_Main_Sound;
        SBB_Main_Sound = new Mac_Sounds();
```

```
//////////////////////////////////////////////
if ( !parse_command ( argc, argv ) )
{
        exit ( 0 );
}
                                        .
                                        .
                                        .
// Initialize Performer
pfInit();
//////////////////////////////////////////
if (include_FLS) FLS_Cyl[0].init_shared();

SBB_Main_Sound->Play_Sound( CHAN1, Good_Morning, 7 );
//////////////////////////////////////////
if ( textures_enabled ) pfFilePath( file_path );
                                        .
                                        .
                                        .
for (i=0; i<3; i++)
   Cyl[i].Coords = ( pfCoord* ) pfMalloc ( sizeof(pfCoord), pfGetSharedArena() );
//////////////////////////////////////////
//// Assign space and initialize the FLS players
if (include_FLS)
{
  FuzzySen.config();
  FLS_Cyl[0].assign_space_and_initialize_FLS_players(
              FuzzySen, FLS_Cyl, &Renderobj, &SimMan,
              &LocalModelMgr);
}
//////////////////////////////////////////////
// Assign a simple terrain for this simulation
appl->Ter = &terrain;
Ov[1].terrain = &terrain;
```

```
                    .
                    .
                    .
        exit(0);
}// end main
```

6. *Compiling and Linking*

There is a main makefile in the SBB/pfmr directory. This make file will create the SBB with the Sentinel extension. There are also two scrip files, MakeAll4 and MakeAll5, that will do a complete cleaning, make depend, recompile, create the necessary Sentinel libraries, compile the SBB application, and link every together. User MakeAll4 with machines that are using the OS4 operating system and MakeAll5 for machines using the OS5 operating system.

Also, inside of each structured program unit's directory, there is a makefile that will compile the library associated with that structured program unit. The makefile will compile the library and place it in the appropriate directory needed by the SBB application's makefile. Have fun!!

BIBLIOGRAPHY

[Air90] Airey, John M.; Rohlf, John H.; and Brooks, Frederick P. Jr. "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments," *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, Snowbird, Utah, pp. 41-50, 25 - 28 March 1990.

[Ali92] Aliev, R.A.; Aliev, F.T.; and Babaev, M.D.. "The Synthesis Of A Fuzzy Coordinate-Parametric Automatic Control System For An Oil-Refinery Unit,"Fuzzy Sets and Systems, Volume 47 Number 2 (April 1992)

[And81] Anderson, J.R. (ed.). *Cognitive Skills and Their Acquisition*, Hillsdale, NJ: Erlbaum., 1981

[App92a] Appino, Perry A.; Lewis, J. Bryan; Ling, Daniel T.; Rabenhorst, David A.; and Codella, Christopher F. "An Architecture for Virtual Worlds," *Presence*, vol. 1, no. 1, pp. 1-17, Winter 1992.

[App92b] Appino, Perry A.; Lewis, J. Bryan; Koved, Lawrence; Ling, Daniel T.; Rabenhorst, David A.; and Codella, Christopher F. "An Architecture for Virtual Worlds," *IBM T.J. Watson Research Center Research Report*, Yorktown Heights, NY, 1992.

[BBN92] Bolt, Beranek and Newman, Inc. *The SIMNET Network and Protocols*. BBN Report no. 7627. 1992.

[Ben82] Benson, William H., *An Application of Fuzzy Set Theory to Data Display*. Fuzzy Set and Possibility Theory Recent Developments, pp. 429-438, 1982.

[Ber93] Bergman, Lawrence D.; Richardson, Jane S.; Richardson, David C.; and Brooks, Frederick P. Jr. "VIEW - An Exploratory Molecular Visualization System with User-Definable Interaction Sequences," *Computer Graphics*, vol. 27, pp. 117-126, August 1993.

[Bes92] Bess, Rick D. "Image Generation Implications for Networked Tactical Training Systems," *Proceedings of the IMAGE VI Conference*, Phoenix, Arizona, pp. 77-86, 14 - 17 July 1992.

[Bla90] Blanchard, Chuck; Burgess, Scott; Harvill, Young; Lanier, Jaron; Lasko, Ann; Oberman, Mark; and Teitel, Michael. "Reality Built for Two: A Virtual Reality Tool," *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, Snowbird, Utah, pp. 35-36, 25 - 28 March 1990.

[Bla92] Blau, Brian; Hughes, Charles E.; Moshell, J. Michael; and Lisle, Curtis. "Networked Virtual Environments," *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, Cambridge, Massachusetts, pp. 157-160, 29 March - 1 April 1992.

[Bla93] Blau, Brian; Moshell, J. Michael; and McDonald, Bruce. "The DIS (Distributed Interactive Simulation) Protocols and their Application to Virtual Environments," *Proceedings of the Meckler Virtual Reality '93 Conference*, San Jose, California, 19 - 21 May 1993.

[Bro86] Brooks, F.P. "Walkthrough - A Dynamic Graphics System for Simulating Virtual Buildings." *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, Chapel Hill, North Carolina, pp. 9-21, 1986.

[Bro88] Brooks, Frederick P. Jr. "Grasping Reality Through Illusion - Interactive Graphics Serving Science," *Human Factors in Computing Systems: CHI '88 Conference Proceedings*, Washington, D.C., pp. 1-11, 15 - 19 May 1988.

[Bro90] Brooks, F.P; Ouh-Young, Ming; and Batter, J. "Project GROPE - Haptic Displays for Scientific Visualization," *Computer Graphics*, vol. 24, no. 4, pp. 177-185, August 1990.

[Bry91] Bryson, Steve and Levit, Creon. "The Virtual Windtunnel: An Environment for the Exploration of Three-Dimensional Unsteady Flows," *Proceedings of Visualization '91*, San Diego, California, pp. 17-24, 22 - 25 October 1991.

[Bry92a] Bryson, Steve and Levit, Creon. "The Virtual Windtunnel: An Environment for the Exploration of Three-Dimensional Computer-Generated Flowfields," *Proceedings of the IMAGE VI Conference*, Phoenix, Arizona, pp. 137-139, 14 - 17 July 1992.

[Bry92b] Bryson, Steve. "Virtual Spacetime: An Environment for the Visualization of Curved Spacetimes via Geodesic Flows," *Proceedings of Visualization '92*, Boston, Massachusetts, pp. 291-298, 19 - 23 October 1992.

[But92] Butterworth, Jeff; Davidson, Andrew; Hench, Stephen; and Olano, T. Marc. "3DM: A Three Dimensional Modeler Using a Head-Mounted Display," *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, Cambridge, Massachusetts, pp. 135-138, 29 March - 1 April 1992.

[Car73] Caso,P.W.; Isley, R.N.; and Jolley, O.B. *Research on Synthetic Training: Device Evaluation and Training Program Development*, (Tech. Report 73-20). Alexandria, VA: Human Resources Research Organization.

[Car92] Carlson, R.A., Lundy, D.H., Schneider, W., "Strategy Guidance and Memory Aiding in Learning a Problem-Solving Skill," *Human Factors*, vol. 34, no. 2, pp. 129-145, April, 1992.

[Chu89] Chung, J.C.; Harris, M.R.; Brooks, F.P.; Fuchs, H.; Kelley, M.T.; Hughes, J.; Ouh-young, M.; Cheung, C.; Holloway, R.L.; and Pique, M. "Exploring Virtual Worlds with Head-Mounted Displays," *Three-Dimensional Visualization and Display Technologies*, SPIE vol. 1083, Los Angeles, California, pp. 42-52, 18-20 January 1989.

[Cod92] Codella, Christopher; Jalili, Reza; Koved, Lawrence; Lewis, J. Bryan; Ling, Daniel T.; Lipscomb, James S.; Rabenhorst, David A.; Wang, Chu P.; Norton, Alan; Sweeney, Paula; Turk, Greg. "Interactive Simulation in a Multi-Person Virtual World," *Human Factors in Computing Systems, SIGCHI '92 Conference Proceedings*, Monterey, California, pp. 329-334, 3 - 7 May 1992.

[Fah93] Fahlen, Lennart E.; Brown, Charles Grant; Stahl, Olov; and Carlsson, Christer. "A Space Based Model for User Interaction in Shared Synthetic Environments," *Conference on Human factors in Computing Systems*, Amsterdam, The Netherlands, pp. 43-50, 24 - 29 April 1993.

[Fal93] Falby, John S.; Zyda, Michael J.; Pratt, David R.; and Mackey, Randy L. "NPSNET: Hierarchical Data Structures for Real-Time Three-Dimensional Visual Simulation," *Computers & Graphics*, vol. 17, no. 1, pp. 65-69. January 1993.

[Fei90a] Feiner, Steven and Beshers, Clifford. "Visualizing *n*-Dimensional Virtual Worlds with *n*-Vision," *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, Snowbird, Utah, pp. 37-38, 25 - 28 March 1990.

[Fei90b] Feiner, Steven and Beshers, Clifford. "Worlds Within Worlds: Metaphors for Exploring n-Dimensional Virtual Worlds," *Proceedings of UIST '90, The Third Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, pp. 76-83, 3 - 5 October 1990.

[Fer92] Ferguson, Robert L.; Brasch, Randy; Lisle, Curtis R.; and Goldiez, Brian. "Interoperability of Visual Simulation Systems," *Proceedings of the IMAGE VI Conference*, Phoenix, Arizona, pp. 517-526, 14 - 17 July 1992.

[Fis86a] Fisher, S.S.; McGreevy, M.; Humphries, J.; and Robinett, W. "A Virtual Environment Display System." *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, Chapel Hill, North Carolina, pp. 77-87, 1986.

[Fis86b] Fisher, S.S.; McGreevy, M.; Humphries, J.; and Robinett, W. "Virtual Workstation: A Multimodal, Stereoscopic Display Environment," *Intelligent Robots and Computer Vision*, SPIE vol. 726, Cambridge, Massachusetts, pp. 517-522, 28 - 31 October 1986.

[Fol90] Foley, James D., et al. *Computer Graphics: Principles and Practice* (2 Edition). Addison-Wesley Publishing Company, 1990.

[Fun92] Funkhouser, T. A., Sequin, C. H., and Teller, S. J. "Management of Large Amounts of Data in Interactive Building Walkthroughs." *1992 Symposium on Interactive 3D Graphics, Computer Graphics*, vol.???, no. ???, pp. 11-20, March 1992.

[Gar92] García, J.; Pazos, J.; Ríos, J.; and Yagüe. "Methodology Of Linguistics Evaluation In Risk Situations Using Fuzzy Techniques," Fuzzy Sets and Systems, Volume 48 Number 2 (June 1992)

[Har91] Harvey, Edward P. and Schaffer, Richard L. "The Capability of the Distributed Interactive Simulation Network Standard to Support High Fidelity Aircraft Simulation." *Proceedings of the Thirteenth Interservice/Industry Training Systems Conference*, Orlando, Florida, pp. 127-135, 1991.

[Hil92] Hill, Ralph D. "The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications," *Human Factors in Computing Systems, SIGCHI '92 Conference Proceedings*, Monterey, California, pp. 335-342, 3 - 7 May 1992.

[Hit92] Hitchner, Lewis E. "Virtual Planetary Exploration: A Very Large Virtual Environment." *Implementation of Immersive Virtual Environments Course Notes, SIGGRAPH 92.*

[Hob93] Hobbs, Bruce A. and Stytz, Martin R. "A User Interface to a True 3-D Display Device," *Proceedings of HCI International '93: 5th International Symposium on Human-Computer Interaction*, Orlando, Florida, pp. ????, 8 - 13 August 1993.

[Kin77] King, P.J. and Mamdani, E.H. "The Application of Fuzzy Control Systems to Industrial Processes," *Automata*, vol. 13, no. 3, pp. 235-242, 1977.

[Kos92] Kosko, Bart. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*, Prentice Hall: Englewood Cliffs, New Jersey, 1992.

[Lee90] Lee, Chuen Chien. "Fuzzy Logic in Control Systems: Fuzzy Logic Controller - Part 1," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 20, no. 2, pp. 404-418, April 1990.

[Lev92] Levit, Creon and Bryson, Steve. "Lessons Learned While Implementing the Virtual Windtunnel Project," *Implementation of Immersive Virtual Environments Course Notes, SIGGRAPH 92.*

[Mam74] Mamdani, E.H. "Application of Fuzzy Algorithms for the Control of a Dynamic Plant," *Proceedings of the IEEE*, vol. 121, no. 12, pp. 1585-1588, 1974.

[McD90] McDonald, L.B., Pinon, C., Glasgow, R., and Danisas, K. "The Standardization of Protocol Data Units for Interoperability of Defense Simulations," *Proceedings of the Twelfth Interservice/Industry Training Systems Conference*, Orlando, Florida, pp. 93-102, 6 - 8 November 1990.

[McD91] McDonald, L. Bruce; Bouwens, Christina P; Hofer, Ronald; Wiehagen, Gene; Danisas, Karen; and Shiflett, James. "Standard Protocol Data Units for Entity Information and Interaction in a Distributed Interactive Simulation." *Proceedings of the Thirteenth Interservice/Industry Training Systems Conference*, Orlando, Florida, pp. 119-126, 1991.

[McL91] McLendon, Patricia. *Graphics Library Programming Guide.* Mountain View, CA: Silicon Graphics, Inc., 1991.

[McL92] McLendon, Patricia. *IRIS Performer Programming Guide*. Mountain View, CA: Silicon Graphics, Inc., 1992.

[Mer90] Mercer, Lynn; Prusinkiewicz, Przemyslaw; and Hanan, James. "The Concept and Design of a Virtual Laboratory," *Graphics Interface '90*, Halifax, Nova Scotia, pp. 149-155, 14 - 18 May 1990.

[Mic88] Michel, R.R. and Reidel, S.L., *Effects of Expertise and Cognitive Style on Information Use in Tactical Decision Making*, (Tech. Report 806), U.S. Army Research Institute, AD-A203 462, June 1988.

[Mil88] Miller, Duncan C.; Pope, Arthur R.; and Waters, Rolland M. "Long-haul Networking of Simulators," *Proceedings of the Eighth Interservice/Industry Training Systems Conference*, Orlando, Florida, pp. 577-582, 1988.

[Min88] Ming, O. Y., Pique, M., Hughes, J., Brooks, Jr., F. P. "Using a Manipulator for Force Display in Molecular Docking." *IEEE Robotics and Automation Conference*. 1988.

[Mos86] Mosher, Charles E. Jr.; Sherouse, George W.; Mills, Peter H.; Novins, Kevin L.; Pizer, Stephen M.; Rosenman, Julian G.; and Chaney, Edward L. "The Virtual Simulator," *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, Chapel Hill, North Carolina, pp. 37-42, 23 - 24 October 1986.

[Ove92] Overmars, Mark H. *Forms Library v2.1 - A Graphical User Interface Toolkit for the Silicon Graphics Workstations*. Utrecht University, Netherlands, Department of Computer Science, 1992. email mmrkov@cs.ruu.nl.

[Pau91] Pausch, Randy. "Virtual Reality on Five Dollars a Day," *Human Factors in Computing Systems: CHI '91 Conference Proceedings*, New Orleans, Louisiana, pp. 265-270, 27 April - 2 May 1991.

[Pic92] Pickover, Clifford A. "A Vacation on Mars - An Artist's Journey in a Computer Graphics World," *Computers & Graphics*, vol. 16, no. 1, pp. 9-13, January 1992.

[Pra92] Pratt, David R.; Zyda, Michael J.; Mackey, Randall L.; and Falby, John S. "NPSNET: A Networked Vehicle Simulation with Hierarchical Data Structures," *Proceedings of the IMAGE VI Conference*, Phoenix, Arizona, pp. 217-225, 14 - 17 July 1992.

[Pra93] Pratt, David R.; Walter, Jon C.; Warren, Patrick T.; and Zyda, Michael J. "NPSNET: Janus-3D Providing Three-Dimensional Displays for a Two-Dimensional Combat Model," *Fourth IEEE Conference on AI, Simulation, and Planning in High Autonomy Systems*, Tucson, Arizona, pp. 31-39, 20 - 22 September 1993.

[Qia92] Qiao, Wu Zhi; Zhuang, Wang Pei; and Heng, Teh Hoon. "A Rule Self-Regulating Fuzzy Controller,"Fuzzy Sets and Systems, Volume 47 Number 1 (April 1992)

[Ram92] Ramakrishnan, R. and Rao, C.J.M.. "The Fuzzy Weighted Additive Rule,"Fuzzy Sets and Systems, Volume 46 Number 2 (March 1992)

[Ras86] Rasmussen, J. *Information Processing and Human-Machine Interaction: An Approach to Cognitive Engineering*. Amsterdam: North Holland, 1986.

[Sch82] Schmandt, Christopher. "Interactive Three-Dimensional Computer Space," *Processing and Display of Three-Dimensional Data*, SPIE vol. 367, San Diego, California, pp. 155-159, 26 - 27 August 1982.

[She92] Sheasby, Steven M. *Management of SIMNET and DIS Entities in Synthetic Environments*. MS thesis, Air Force Institute of Technology, 1992.

[Smi91] Smith, Bradford. "The Use of Animation to Analyze and Present Information About Complex Systems," *Proceedings of Virtual Reality '91, The Second Annual Conference on Virtual Reality, Artificial Reality, and Cyberspace*, San Francisco, California, pp. 190-199, 23 - 25 September 1991.

[Sny93] Snyder, Mark. *ObjectSim Framework ???????*. MS thesis, Air Force Institute of Technology, 1993.

[Sol91] Solso, R.L., *Cognitive Psychology*, 3rd ed., Allyn and Bacon, 1991.

[Sol92] Soltz, Brian B., *Macintosh Sound Generation Facility 2.0*. Special Study, Air Force Institute of Technology, 1992.

[Tay93] Taylor, Russell M.; Robinett, Warren; Chi, Vernon L.; Brooks, Frederick P. Jr.; Wright, William V.; Williams, R. Stanley; and Snyder, Eric J. "The Nanomanipulator: A Virtual-Reality Interface for a Scanning Tunneling Microscope," *Computer Graphics*, vol. 27, pp. 127-134, August 1993.

[Tel91] Teller, S. J. and Sequin, C. H. "Visibility Preprocessing For Interactive Walkthroughs." *Computer Graphics*, vol. 25, no. ???, pp. ?????, ?????, August 1991.

[Tho31] Thorndike, E.L. *Human Learning*. New York: Century, 1931.

[Tho88] Thorpe, Jack. "Warfighting with SIMNET - A Report From the Front," *Proceedings of the 10th Interservice/Industry Training Systems Conference*, Orlando, Florida, pp. 263-273, 1988.

[Was72] Wason, P.C. and Johnson-Laird, P.N. *Psychology of Reasoning: Structure and Content*. London: Batsford, 1972.

[Wei89] Weimer, David and Ganapathy, S.K. "A Synthetic Visual Environment with Hand Gesturing and Voice Input," *Human Factors in Computing Systems: CHI '89 Conference Proceedings*, Austin, Texas, pp. 235-240, 30 April - 3 May 1989.

[Wil93] Wilson, Kirk. *Synthetic Battle Bridge ???????*. MS thesis, Air Force Institute of Technology, 1993.

[Zad73] Zadeh, Lofti A. "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-3, no. 1, pp. 28-44, January 1973.

[Zel92] Zeltzer, David and Drucker, Steven. "A Virtual Environment System for Mission Planning," *Proceedings of the IMAGE VI Conference*, Phoenix, Arizona, pp. 125-134, 14 - 17 July 1992.

[Zyd92] Zyda, Michael J.; Pratt, David R.; Monahan, James G.; and Wilson, Kalin P. "NPSNET: Constructing a 3D Virtual World," *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, Cambridge, Massachusetts, pp. 147-156, 29 March - 1 April 1992.

VITA

Brian B. Soltz was born on May 12, 1961, in Atlantic City, New Jersey. On March 15, 1982, Brian entered the United States Air Force as a Morse Code Systems Operator (208X0). In March, 1983, Brian became a instructor in the Morse Systems Training School at Keelser AFB, Mississippi. In April 1984, Brian was accepted by the Airman's Education Commissioning Program (AECP) and sent to Rutgers University. In May 1988, Brian received his Bachelor of Science in Electrical Engineering from Rutgers University.

In September 1988, Brian was commissioned a second lieutenant in the Air Force through the Officer Training School at Lackland AFB, Texas. In September 1988, Brian was stationed at Wright-Patterson AFB, Ohio as a C^3 Systems Analyst. In 1992, Brian was accepted to the Air Force Institute of Technology, where he completed his Master of Science degree in Computer Science in 1993.

December 1993 Master's Thesis

GRAPHICAL TOOLS FOR SITUATIONAL AWARENESS ASSISTANCE FOR LARGE SYNTHETIC BATTLE SPACES

Brian B. Soltz, Capt, USAF

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GCS/ENG/93D-21

ARPA/ASTO
3701 North Fairfax Drive
Arlington, Va 22203

As virtual environments grow in complexity and size, users are increasingly challenged in assessing situations in large-scale virtual environment. This occurs because of the difficulty in determining where to focus attention and assimilating and assessing the information as it floods in. One technique for providing this type of assistance is to provide the user with a first-person, immersive, synthetic environment observation post, that permits unobtrusive observation of the environment without interfering with the activity in the environment. However, for large, complex synthetic environments, this type of support is not sufficient because the portrayal of raw, unanalyzed data in the virtual space can overwhelm the user. To address these problems, this thesis investigates the types of situational awareness assistance that needs to be provided to users of large-scale virtual environments. A technique developed, is to allow a user to place analysis modules throughout the virtual environment. Each module provides summary information to the user concerning the status of the section of the virtual environment that the module was assigned to monitor. The prototype system, called the Sentinel, is embedded within a virtual environment observatory and provides situational awareness assistance for users within a large virtual environment.

Fuzzy Logic, Situational Awareness, Synthetic Environments, Object-Oriented, Computer Graphics

184

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL